



**POLITECHNIKA LUBELSKA
WYDZIAŁ ELEKTROTECHNIKI
I INFORMATYKI**

**KIERUNEK STUDIÓW
INFORMATYKA**

***MATERIAŁY DO ZAJĘĆ
LABORATORYJNYCH***

Zaawansowane programowanie w Javie

Autor:
mgr inż. Piotr Wójcicki

Lublin 2022



**Fundusze
Europejskie**
Wiedza Edukacja Rozwój



**Rzeczpospolita
Polska**

Unia Europejska
Europejski Fundusz Społeczny



INFORMACJA O PRZEDMIOCIE

Cele przedmiotu:

- Cel 1. Zapoznanie studentów z zaawansowanymi możliwościami języka Java
- Cel 2. Nabycie umiejętności przez studentów pisania zaawansowanych aplikacji w języku Java
- Cel 3. Zapoznanie studentów z najważniejszymi bibliotekami wykorzystywanymi przez programistów Javy

Efekty kształcenia w zakresie umiejętności:

- Efekt 1. Potrafi programować aplikacje w języku Java, korzystając z wybranych wzorców projektowych i zaawansowanych elementów języka
- Efekt 2. Potrafi umiejętnie stosować wybrane biblioteki języka Java w celu efektywnego tworzenia aplikacji w zależności od zastosowania

Literatura do zajęć:

Literatura podstawowa

1. Bloch J., Java. Efektywne programowanie. Wydanie III, Helion, Gliwice 2018
2. Horstmann, C.S., Java. Techniki zaawansowane. Wydanie X, Helion, Gliwice 2017
3. Krawiec J., Java. Programowanie obiektowe w praktyce, Oficyna Wydawnicza Politechniki Warszawskiej, Warszawa 2017
4. Smart J. F, Java. Praktyczne narzędzia. Helion, Gliwice 2009

Literatura uzupełniająca

1. Eckel B., Thinking in Java, Wydanie IV, Helion, Gliwice 2006
2. Ganeshan A., Spring MVC. Przewodnik dla początkujących, Helion, Gliwice 2015
3. Gutierrez F., Wprowadzenie do Spring Framework dla programistów Java, Helion, Gliwice 2015
4. Bauer C., King G., Gregory G., Java Persistence. Programowanie aplikacji bazodanowych w Hibernate, Helion, Gliwice 2017
5. Rocha R., Purificacao J., Java EE 8. Wzorce projektowe i najlepsze praktyki, Helion, Gliwice 2019
6. Warin G., Spring MVC 4. Projektowanie zaawansowanych aplikacji WWW, Helion, Gliwice 2016
7. Dokumentacja Spring Framework [online] <https://docs.spring.io/spring-framework/>
8. Javapoint [online] <https://www.javatpoint.com>
9. Javappa [online] <https://javappa.com>
10. Baelung [online] <https://www.baeldung.com>
11. Edencoding [online] <https://edencoding.com>

Metody i kryteria oceny:

- Oceny częściowe:
 - Ocena 1 Przygotowanie merytoryczne do zajęć laboratoryjnych na podstawie: wykładów, literatury, pytań kontrolnych do zajęć.
 - Ocena 2 Zaliczenie na podstawie oddanych projektów.
- Ocena końcowa - zaliczenie przedmiotu:
 - Pozytywne oceny częściowe.
 - Ewentualne dodatkowe wymagania prowadzącego zajęcia.



Fundusze Europejskie
Wiedza Edukacja Rozwój



**Rzeczpospolita
Polska**

Unia Europejska
Europejski Fundusz Społeczny



Plan zajęć laboratoryjnych:

Lab1.	Laboratorium 1.
Lab2.	Laboratorium 2. Wykorzystanie plików XML.
Lab3.	Laboratorium 3. Tworzenie aplikacji sieciowych.
Lab4.	Laboratorium 4. Tworzenie aplikacji bazodanowych.
Lab5.	Laboratorium 5. Internacjonalizacja w języku Java.
Lab6.	Laboratorium 6. Tworzenie aplikacji z wykorzystaniem biblioteki Lombok.
Lab7.	Laboratorium 7. Tworzenie aplikacji z wykorzystaniem biblioteki JavaFX.
Lab8.	Laboratorium 8. Tworzenie aplikacji z wykorzystaniem biblioteki Spring.
Lab9.	Laboratorium 9. Tworzenie aplikacji z wykorzystaniem biblioteki Hibernate.
Lab10.	Laboratorium 10. Tworzenie aplikacji z wykorzystaniem biblioteki JSoup.
Lab11.	Laboratorium 11. Wybrane zagadnienia dotyczące bezpieczeństwa.
Lab12.	Laboratorium 12. Testy jednostkowe JUnit.

Sugerowana kolejność wykonywania laboratoriów:

1 – 2 – 6 – 7 – 8 – 9 – 10 – 11 – 12 – 5 – 4 – 3



Fundusze Europejskie
Wiedza Edukacja Rozwój



**Rzeczpospolita
Polska**

Unia Europejska
Europejski Fundusz Społeczny



LABORATORIUM 1. STRUMIENIE I PLIKI

Cel laboratorium:

Zapoznanie z pojęciem strumienia, operacjami na strumieniach. Obsługa plików.

Zakres tematyczny zajęć:

- Strumień, operacje na strumieniach.
- Obsługa plików.

Pytania kontrolne:

1. Czym jest strumień?
2. W jaki sposób możliwe jest utworzenie strumienia? Co jest niezbędne?
3. Do czego wykorzystuje się strumień?
4. Jakie znasz operacje wykonywane na strumieniach?
5. W jaki sposób należy zapisać dane do pliku?
6. W jaki sposób należy odczytać dane z pliku?
7. Czy źródłem danych dla strumienia mogą być pliki?

Strumień

Strumień (ang. stream) jest abstrakcyjnym pojęciem reprezentującym dowolne źródło wejściowe lub wyjściowe danych jako obiektu zdolnego do wysyłania lub odbierania porcji danych. To co dzieje się z danymi wewnątrz urządzenia wejścia – wyjścia jest ukryte. W języku Java biblioteki wejścia – wyjścia są podzielone według wejścia (ang. input) oraz wyjścia (ang. output). Wykorzystując mechanizm dziedziczenia wszystkie klasy rozszerzone przy użyciu klas `InputStream` lub `Reader` mają metody odczytu jednego bajta lub tablicy bajtów – `read()`. Analogicznie klasy rozszerzające `OutputStream` lub `Writer` posiadają metody do zapisu jednego bajta lub tablicy bajtów – `write()`.

W rzeczywistości nie zachodzi potrzeba użycia podanych wyżej metod bezpośrednio – wykorzystują je klasy podrzędne dostarczając jednocześnie bardziej użyteczny interfejs. Obiekt strumieniowy tworzy się zazwyczaj poprzez nawarstwienie kilku obiektów różnych klas. Pomocny jest tutaj podział na kategorie wejścia – dziedziczące po `InputStream` i wyjścia – dziedziczące po `OutputStream`.

Klas pochodnych po wyżej wymienionych jest wiele i w zależności od źródła danych należy wykorzystać odpowiednie z nich. Lista oficjalnych klas wykorzystujących `InputStream` lub `OutputStream` jest dostępna w dokumentacji na pozycji znane podklasy (Direct Known Subclasses).

Podstawową klasą interfejsu API jest `Stream<T>`. Do stworzenia strumienia można wykorzystać różne źródła elementów takich jak kolekcje lub tablice stosując metodę `stream()` oraz `of()`. Przykładowy kod pozwalający na utworzenie strumienia ze źródła danych jakim jest tablica został przedstawiony poniżej. Elementami tablicy mogą być składowe dowolnego typu prostego lub klasy.

```
String[] arr = new String[]{"alfa", "beta", "gamma"};
Stream<String> stream = Arrays.stream(arr);
```



```
stream = Stream.of("a", "b", "c");
```

W przypadku użycia dowolnej kolekcji jako źródłowego elementu dla strumienia do interfejsu `Collection` jest dodawana domyślna metoda `stream()`, która pozwala na tworzenie strumieni. Dzięki temu dowolne źródło np. w postaci listy może zostać bezpośrednio użyte do tworzenia strumienia.

```
Stream<String> stream = list.stream();
```

Kolejną ważną cechą współpracy API stream z pozostałymi elementami języka Java jest udział w projektach korzystających z pracy wielowątkowej. API strumieni pozwala na wykorzystanie metody `parallelStream()`, która startuje operacje na elementach strumienia ale w trybie równoległym.

```
list.parallelStream().forEach(element -> doWork(element));
```

Operacje na strumieniach można podzielić na dwa rodzaje. Pierwszy rodzaj (ang. intermediate operations) zwraca strumień – jeżeli jako dane źródłowe został przekazany łańcuch znaków to jako wynik takiej operacji zostanie zwrócony `Stream<String>`. Drugi rodzaj to operacje pośrednie (ang. terminal operations) i wynikiem działania takiej operacji może być konkretny typ np. `String`. Bardzo istotne jest zaznaczenie, że operacje na strumieniach nie zmieniają stanu czy zawartości źródła. Przykładem połączenia wspomnianych operacji może być poniższy przykład. Metoda `count()` zwraca rozmiar strumienia i jest zaliczana do operacji drugiego rodzaju (dowolny typ), natomiast `distinct()` tworzy z istniejącego nowy strumień bez ingerencji w pierwowzór.

```
int count = list.stream().distinct().count();
```

Dużą zaletą stosowania interfejsu strumieni jest możliwość iterowania po elementach. Dzięki temu możliwe jest pominięcie w wielu przypadkach stosowania pętli na rzecz ciekawszych rzeczy, jak na przykład logika biznesowa. Poniższy kod przedstawia podejście tradycyjne (z użyciem pętli) oraz wykorzystujące dobrodziejstwa API.

```
for (String string : list) {  
    if (string.contains("a")) {  
        return true;  
    }  
}
```

Ten sam efekt można uzyskać przy pomocy poniższego kodu.

```
boolean isExist = list.stream().anyMatch(element ->  
element.contains("a"));
```

Operacje filtrowania są niezwykle użyteczne – wystarczy, że elementy przeszukiwanego zbioru spełniają predykat i wówczas są oddzielane od pozostałych. Przykład kodu pozwalającego na filtrowanie strumienia został przedstawiony poniżej.

```
ArrayList<String> list = new ArrayList<>();  
list.add("Today");  
list.add("TestIxD");  
list.add("Java");  
list.add("Tomorrow");  
list.add("JavaFX");  
list.add("joke");  
list.add("noMercy");  
list.add("for");  
list.add("lazy");  
list.add("students");  
list.add("");  
list.add("");
```

Dla tak przygotowanej listy `ArrayList<String>` tworzony jest strumień postaci `Stream<String>`, po którym następuje filtrowanie i wszystkie elementy spełniające predykat (w tym przypadku zawierają element „xD”) są brane jako źródło dla nowego strumienia. W efekcie nowo utworzony strumień będzie zawierał tylko szukane elementy.

```
Stream<String> stream = list.stream().filter(element ->  
element.contains("xD"));
```

Do wyświetlenia można wykorzystać wyrażenie lambda oraz iterację po elementach strumienia.

```
stream.forEach(s -> System.out.println(s));
```

Lub pomijając wyrażenie lambda i niepotrzebny parametr `s`:

```
stream.forEach(System.out::println);
```

Inne przydane operacje to mapowanie (ang. `mapping`), dopasowywanie (ang. `matching`) oraz redukowanie (ang. `reduction`).

Zapis i odczyt plików

Zapis do pliku może być zrealizowany z wykorzystaniem klasy `FileWriter` jak pokazano na poniższym przykładzie. Należy zwrócić uwagę na konieczność obsługi wyjątku klasy `IOException`.

```
public static void prostyZapis() {  
  
    File file = new File("WynikZapisu.txt");  
    try {  
        FileWriter fileWriter = new FileWriter(file,  
true);  
        for (int i = 0; i < 10; i++) {
```



```
        fileWriter.append("To jest dopisana linijka  
tekstu nr: " + i + "\n");  
    }  
    fileWriter.close();  
} catch (IOException ex) {  
    System.err.println(ex.getCause());  
}  
}
```

Najpierw tworzona jest instancja klasy File (można pominąć to i niejawnie utworzyć obiekt File w konstruktorze FileWriter). Następnym krokiem jest wyłapanie ewentualnych wyjątków blokiem try – catch (IOException). Tworzony obiekt FileWriter posiada argument wskazujący na plik oraz opcjonalnie wartość true oznaczającą możliwość dopisywania kolejnych linii do pliku – drugie uruchomienie metody prostyZapis() dopisze do już istniejącego pliku ciąg znaków (bez drugiego argumentu plik zostałby nadpisany od początku). Po pętli dopisującej kolejne linie następuje zamknięcie strumienia – należy zawsze o tym pamiętać. Zmienić się może miejsce wywołania metody close() w zależności od potrzeb.

Odczyt z pliku również może być realizowany przy użyciu wielu klas. FileReader oferuje dostęp znak po znaku, BufferedReader linijka po linijce. Przy tworzeniu instancji parametrami konstruktora mogą być obiekty tworzone na bieżąco jak w poniższym przykładzie.

```
public static void prostyOdczyt() {  
  
    try {  
        BufferedReader bufferedReader = new  
BufferedReader(new FileReader(new File("WynikZapisu.txt")));  
        String linia = null;  
  
        while ((linia = bufferedReader.readLine()) !=  
null) {  
            System.out.println(linia);  
        }  
        bufferedReader.close();  
    } catch (FileNotFoundException ex) {  
        System.out.println("Pliku nie odnaleziono!");  
        System.err.println(ex.getCause());  
    } catch (IOException ex) {  
        System.out.println("Błąd odczytu pliku  
spowodowany:");  
        System.err.println(ex.getCause());  
    }  
}
```

Odczyt linijka po linijce najprościej zaimplementować przy użyciu klasy BufferedReader. Przy otwieraniu pliku należy zawsze uwzględnić jego nieistnienie lub błąd



w podawanej nazwie (literówka) dlatego należy wyłapywać wyjątek klasy `FileNotFoundException`. W dalszej części w pętli `while` która działa dopóki plik nie zakończy się (koniec pliku oznaczony jest nullem) odczytywana jest każda kolejna linijka. Po odczycie obowiązkowo należy zamknąć strumień danych.

Przedstawione przykłady nie są jedynymi słusznymi. Istnieje wiele klas przy użyciu których możliwe jest odczytanie pliku (np. `Scanner`) i należy korzystać z nich w zależności od sytuacji.

Zadanie 1.1. Operacje na strumieniach

Utwórz strumień składający się z listy łańcuchów znaków. Lista ma zawierać nazwy wszystkich przedmiotów z zeszłego i obecnego semestru.

- Wykorzystaj funkcję filtrującą do sprawdzenia, czy w zadanym strumieniu nie znajdują się łańcuchy zawierające fragment „zaaw”.
- Zastosuj operację mapowania do utworzenia nowego strumienia. Nowy strumień powinien zawierać listę ale ocen z przedmiotów, które znajdowały się na poprzedniej liście. Dopasuj odpowiedni typ dla nowego strumienia.
- Sprawdź czy w nowo utworzonym strumieniu jakieś oceny się powtarzają i ile razy.

Zadanie 1.2. Operacje na plikach

Wykorzystaj utworzone w zadaniu 1.1 strumienie do zapisu odpowiednich informacji do plików. Należy w taki sposób uporządkować dane, aby w pliku pojawiły się jako pierwsze informacje o przedmiotach, z których oceny były:

- najniższe (kolejność rosnąca),
- najrzadziej się powtarzały (kolejność rosnąca).

Zadanie 1.3. Operacje na plikach

Wykorzystaj plik utworzony w zadaniu 1.2 i odczytaj zawarte w nim informacje. Po wczytaniu danych należy obliczyć średnią ocen za poprzedni i obecny semestr, a następnie wypisać przedmiot z najlepszą i najgorszą oceną.

LABORATORIUM 2. WYKORZYSTANIE PLIKÓW XML.

Cel laboratorium:

Zapoznanie z budową i użyciem plików XML w Javie.

Zakres tematyczny zajęć:

- Budowa pliku XML.
- Zastosowanie plików XML w projektach Java.

Pytania kontrolne:

1. Jakie są cechy charakterystyczne plików XML?
2. W jaki sposób można przekazać istotne informacje (np. o bibliotece) do projektu Java?
3. Dlaczego pliki XML są stosowane do przechowywania właściwości projektu, np. w przypadku aplikacji bazodanowych?
4. Jakie dane można przechowywać w plikach XML?
5. Czy Java jest w jakiś sposób wyposażona w narzędzia do pracy z plikami XML?

Notacja XML

XML, czyli eXtensible Markup Language (rozszerzalny język znaczników, zaprojektowany przez World Wide Web Consortium) jest specyfikacją zapisu danych. Precyzyjnie jest to uniwersalny język znaczników przeznaczony do reprezentowania różnych danych w strukturalizowany sposób. To język znaczników i format pliku do przechowywania, przesyłania i rekonstrukcji dowolnych danych. Jest niezależny od platformy, co umożliwia łatwą wymianę dokumentów pomiędzy heterogenicznymi (różnymi) systemami i znacząco przyczyniło się do popularności tego języka w dobie Internetu. XML jest standardem rekomendowanym oraz specyfikowanym przez organizację W3C. Jest najpopularniejszym obecnie uniwersalnym językiem przeznaczonym do reprezentowania danych. Dokument XML składa się z zagnieżdżonych elementów w postaci znaczników `<znacznik></znacznik>`. Pomiędzy znacznikami umieszczane są elementy. Znacznik rozpoczyna lewy nawias kątowny (`<`), a kończy prawy (`>`). Zapisane elementy w dokumencie przypominają strukturę drzewa, w którym może wystąpić jeden główny element. Prosty dokument w notacji XML został przedstawiony poniżej:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>org.example</groupId>
  <artifactId>LabSecurity</artifactId>
```



```
<version>1.0-SNAPSHOT</version>

</project>
```

Jest to nic innego jak plik pom.xml występujący w każdym projekcie Java zbudowanym na Maven'ie. Część górną stanowi tzw. prolog, natomiast po nim następuje główny element – projekt. Prolog musi występować w każdym dokumencie i może znaleźć się tutaj deklaracja np. na temat sposobu kodowania - `encoding="UTF-8"?` lub odwołania do innych dokumentów (w innych przykładach). Obowiązkowe jest natomiast określenie wersji poprzez `?xml version="1.0"`. Główny element stanowi nazwa (ciągła, bez spacji) i odnosi się do zawartości projektu – w tym przypadku spis podstawowych informacji jak np. grupa, artefakt czy wersja.

Poza oczywistym wykorzystaniem plików XML w projektach Maven'owych służą one przede wszystkim do gromadzenia i porządkowania danych. Przykładowy plik XML został przedstawiony poniżej

(źródło: <https://docs.oracle.com/javase/tutorial/jaxp/stax/example.html#bnbnf>).

```
<?xml version="1.0" encoding="UTF-8"?>
<BookCatalogue xmlns="http://www.publishing.org">
<Book>
  <Title>Yogasana Vijnana: the Science of Yoga</Title>
  <author>Dhirendra Brahmachari</Author>
  <Date>1966</Date>
  <ISBN>81-40-34319-4</ISBN>
  <Publisher>Dhirendra Yoga Publications</Publisher>
  <Cost currency="INR">11.50</Cost>
</Book>
<Book>
  <Title>The First and Last Freedom</Title>
  <Author>J. Krishnamurti</Author>
  <Date>1954</Date>
  <ISBN>0-06-064831-7</ISBN>
  <Publisher>Harper & Row</Publisher>
  <Cost currency="USD">2.95</Cost>
</Book>
</BookCatalogue>
```

Jest to prosty katalog książek, oparty na wspólnej przestrzeni nazw BookCatalogue. W poniższym przykładzie przedstawiono prostą aplikację do odczytu zawartości XML.

```
try {
    for (int i = 0 ; i < count ; i++) {
        // pass the file name.. all relative entity
        // references will be resolved against this
        // as base URI.
        XMLStreamReader xmlr =
xmlif.createXMLStreamReader(filename,
```



```
new
FileInputStream(filename));

    // when XMLStreamReader is created,
    // it is positioned at START_DOCUMENT event.
    int eventType = xmlr.getEventType();
    printEventType(eventType);
    printStartDocument(xmlr);

    // check if there are more events
    // in the input stream
    while(xmlr.hasNext()) {
        eventType = xmlr.next();
        printEventType(eventType);

        // these functions print the information
        // about the particular event by calling
        // the relevant function
        printStartElement(xmlr);
        printEndElement(xmlr);
        printText(xmlr);
        printPIData(xmlr);
        printComment(xmlr);
    }
}
```

Metoda next() zwraca tylko liczby całkowite odpowiadające bazowym typom zdarzeń, zazwyczaj należy zmapować te liczby całkowite na ciągi reprezentujące zdarzenia. Przykład takiego kodu przedstawiono poniżej:

```
public final static String getEventTypeString(int eventType) {
    switch (eventType) {
        case XMLEvent.START_ELEMENT:
            return "START_ELEMENT";

        case XMLEvent.END_ELEMENT:
            return "END_ELEMENT";

        case XMLEvent.PROCESSING_INSTRUCTION:
            return "PROCESSING_INSTRUCTION";

        case XMLEvent.CHARACTERS:
            return "CHARACTERS";

        case XMLEvent.COMMENT:
            return "COMMENT";

        case XMLEvent.START_DOCUMENT:
            return "START_DOCUMENT";
    }
}
```



```
        return "START_DOCUMENT";

    case XMLEvent.END_DOCUMENT:
        return "END_DOCUMENT";

    case XMLEvent.ENTITY_REFERENCE:
        return "ENTITY_REFERENCE";

    case XMLEvent.ATTRIBUTE:
        return "ATTRIBUTE";

    case XMLEvent.DTD:
        return "DTD";

    case XMLEvent.CDATA:
        return "CDATA";

    case XMLEvent.SPACE:
        return "SPACE";
    }
    return "UNKNOWN_EVENT_TYPE , " + eventType;
}
```

Zadanie 2.1. Tworzenie XML

Stwórz klasę modelową (np. Student, Osoba, Książka...) wyposażoną w zestaw pól i metod do ich pobierania i ustawiania. Napisz aplikację do tworzenia plików XML przechowujących dane pobrane z listy/kolekcji/innego źródła dotyczących obiektów klasy modelowej. Dane mają być przechowywane w sposób zgodny z notacją XML.

Zadanie 2.2. Odczyt XML

Wykorzystaj aplikację z zadania 2.1. Napisz program do odczytu danych z plików XML i parsowania ich na obiekt konkretnej klasy.

Zadanie 2.3. Filtracja XML

Wykorzystaj aplikację z zadania 2.2. Napisz program do odczytu wybranych danych z plików XML. Jeżeli dane które są w pliku nie spełniają podstawy do utworzenia obiektu (np. ze względu na konstruktor) aplikacja powinna zwrócić odpowiedni komunikat o braku wymaganych danych.

LABORATORIUM 3. TWORZENIE APLIKACJI SIECIOWYCH.

Cel laboratorium:

Praktyczne tworzenie aplikacji sieciowej.

Zakres tematyczny zajęć:

- Tworzenie połączenia, ustawienia sieciowe.
- Tworzenie aplikacji bez wsparcia frameworków.

Pytania kontrolne:

1. W jaki sposób należy skonfigurować połączenie sieciowe?
2. Co jest niezbędne do otwarcia połączenia?
3. Czy możliwe jest uruchomienie aplikacji serwera na tym samym komputerze, co klient? Z jakich narzędzi należy skorzystać?
4. Jakie znasz biblioteki/frameworki wspomagające pracę nad aplikacjami sieciowymi?

Ustanowienie połączenia

Nawiązanie połączenia z innym komputerem jest możliwe przy użyciu tzw. gniazda. W języku Java gniazdo jest reprezentowane przez obiekt klasy Socket z pakietu java.net. Połączenie sieciowe między dwoma komputerami jest relacją, dzięki której dwa programy wiedzą o swoim istnieniu – mogą się wzajemnie komunikować (przesyłać informacje lub po prostu bity). Do ustanowienia połączenia sieciowego niezbędne są dwie informacje – pierwsza to adres IP, a druga numer portu TCP.

Komputer pełniący rolę serwera dysponuje 65536 portami, z których (prawie) każdy być wykorzystywany przez aplikację pełniącą rolę serwera. Prawie każdy, ponieważ porty TCP od 0 do 1023 są zarezerwowane dla usług powszechnie używanych (np. HTTP – 80, HTTPS – 443, SMTP – 25, POP3 – 110). Dla własnych serwerów rekomendowane jest wybranie portu z zakresu 1024 – 65535.

Komunikator sieciowy

Dla lepszego zobrazowania w jaki sposób działa komunikacja sieciowa przygotowano przykład. Aplikacja – komunikator sieciowy składa się z pola tekstowego, przycisku i wielowierszowego pola tekstowego. W projekcie utworzono dwie klasy – jedna odpowiedzialna za część kliencką (Klient), a druga za część serwerową (Serwer). Aplikacja jest skonfigurowana w taki sposób aby można było uruchomić obie części na jednym komputerze (wykorzystując localhost). Jako pierwsza powinna zostać uruchomiona część serwerowa – należy PPM kliknąć na klasę na drzewie projektowym, a następnie wybrać Run File (ewentualnie zaznaczyć klasę i wykorzystać skrót Shift + F6). Analogicznie należy uruchomić część kliencką. Zapoznaj się z klasą Klient:

```
package com.mycompany.komunikacjasieciowa;
```



Fundusze Europejskie
Wiedza Edukacja Rozwój



**Rzeczpospolita
Polska**

Unia Europejska
Europejski Fundusz Społeczny



```
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.net.*;
import javax.swing.*;

public class Klient {

    JTextArea odbiorWiadomosci;
    JTextField wiadomosc;
    BufferedReader czytelnik;
    PrintWriter pisarz;
    Socket gniazdo;

    public static void main(String[] args) {

        Klient klient = new Klient();
        klient.polaczMnie();
    }

    public void polaczMnie() {
        JFrame frame = new JFrame("Prosty klient czatu");
        JPanel panel = new JPanel();

        odbiorWiadomosci = new JTextArea(15, 50);
        odbiorWiadomosci.setLineWrap(true);
        odbiorWiadomosci.setWrapStyleWord(true);
        odbiorWiadomosci.setEditable(false);

        JScrollPane przewijanie = new
        JScrollPane(odbiorWiadomosci);

        przewijanie.setVerticalScrollBarPolicy(ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);

        przewijanie.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);

        wiadomosc = new JTextField(20);

        JButton przyciskWyslij = new JButton("Wyslij");
        przyciskWyslij.addActionListener(new
        SluchaczPrzycisku());

        panel.add(przewijanie);
        panel.add(wiadomosc);
        panel.add(przyciskWyslij);
        konfiguruje();
    }
}
```



```
Thread watekOdbiorcy = new Thread(new Odbiorca());
watekOdbiorcy.start();

frame.getContentPane().add(BorderLayout.CENTER,
panel);
frame.setSize(new Dimension(600, 400));
frame.setVisible(true);
}

private void konfiguruj() {
    try {
        gniazdo = new Socket("127.0.0.1", 2020);
        InputStreamReader czytelnikStrm = new
InputStreamReader(gniazdo.getInputStream());
        czytelnik = new BufferedReader(czytelnikStrm);
        pisarz = new
PrintWriter(gniazdo.getOutputStream());
        System.out.println("Zakończono konfiguracje
sieci");
    } catch (IOException ex) {
        System.out.println("Konfiguracja sieci nie
powiodła się!");
        ex.printStackTrace();
    }
}

private class SluchaczPrzycisku implements ActionListener
{

    @Override
    public void actionPerformed(ActionEvent e) {
        try {
            pisarz.println(wiadomosc.getText());
            pisarz.flush();
        } catch (Exception ex) {
            ex.printStackTrace();
        }
        wiadomosc.setText("");
        wiadomosc.requestFocus();
    }
}

public class Odbiorca implements Runnable {

    @Override
    public void run() {
        String wiad;
        try {
```




```
        while ((wiad = czytnik.readLine()) != null)
        {
            System.out.println("Odczytano: " + wiad);
            odbiorWiadomosci.append(wiad + "\n");
        }
    } catch (Exception ex) {
        ex.printStackTrace();
    }
}
}
```

Zapoznaj się z klasą Serwer:

```
package com.mycompany.komunikacjasieciowa;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.ArrayList;
import java.util.Iterator;

public class Serwer {

    ArrayList strumienieWyjsciowe;

    public class ObslugaKlienta implements Runnable {

        BufferedReader czytnik;
        Socket gniazdo;

        public ObslugaKlienta(Socket gniazdo) {
            try {
                this.gniazdo = gniazdo;
                InputStreamReader reader = new
InputStreamReader(gniazdo.getInputStream());
                czytnik = new BufferedReader(reader);

                } catch (Exception ex) {
                    ex.printStackTrace();
                }
            }

            @Override
            public void run() {
                String wiadomosc;
                try {
```

```
        while ((wiadomosc = czytnik.readLine()) !=
null) {
            System.out.println("Odczytano: " +
wiadomosc);
            rozeslijDoWszystkich(wiadomosc);
        }
    } catch (IOException ex) {
        ex.printStackTrace();
    }
}

public static void main(String[] args) {
    new Serwer().doRoboty();
}

public void doRoboty() {
    strumienieWyjsciowe = new ArrayList();
    try {
        ServerSocket gniazdoSerwera = new
ServerSocket(2020);

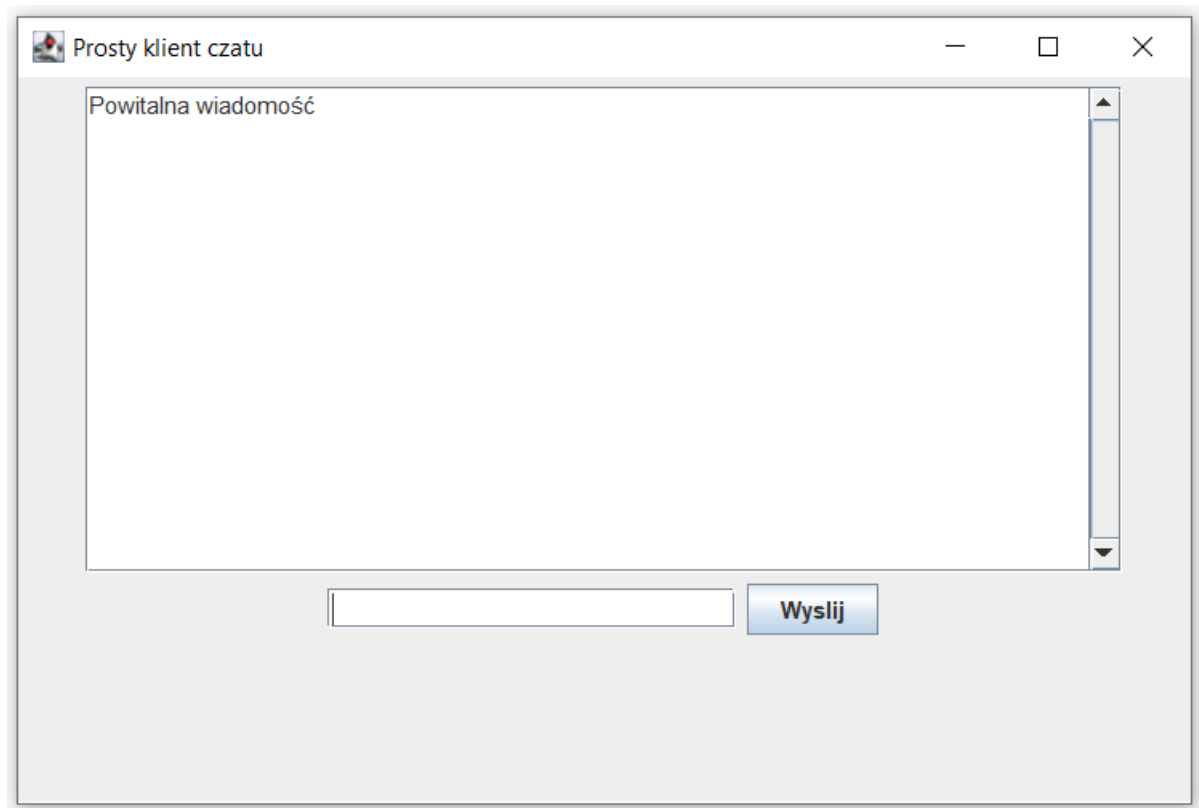
        while (true) {
            Socket gniazdoKlienta =
gniazdoSerwera.accept();
            PrintWriter pisarz = new
PrintWriter(gniazdoKlienta.getOutputStream());
            strumienieWyjsciowe.add(pisarz);

            Thread watekKlienta = new Thread(new
ObslugaKlienta(gniazdoKlienta));
            watekKlienta.start();
            System.out.println("Nawiązano połączenie!");
        }
    } catch (Exception ex) {
        ex.printStackTrace();
    }
}

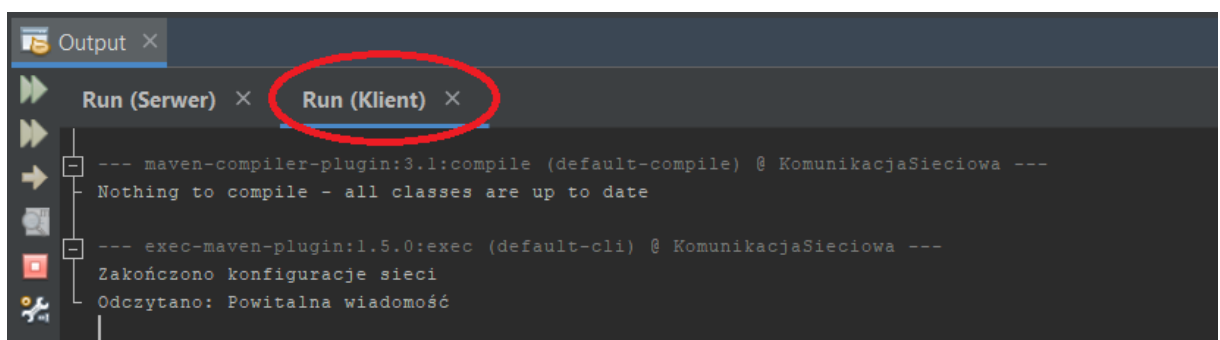
public void rozeslijDoWszystkich(String wiadomosc) {
    Iterator it = strumienieWyjsciowe.iterator();
    while (it.hasNext()) {
        try {
            PrintWriter pisarz = (PrintWriter) it.next();
            pisarz.println(wiadomosc);
            pisarz.flush();
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}
```



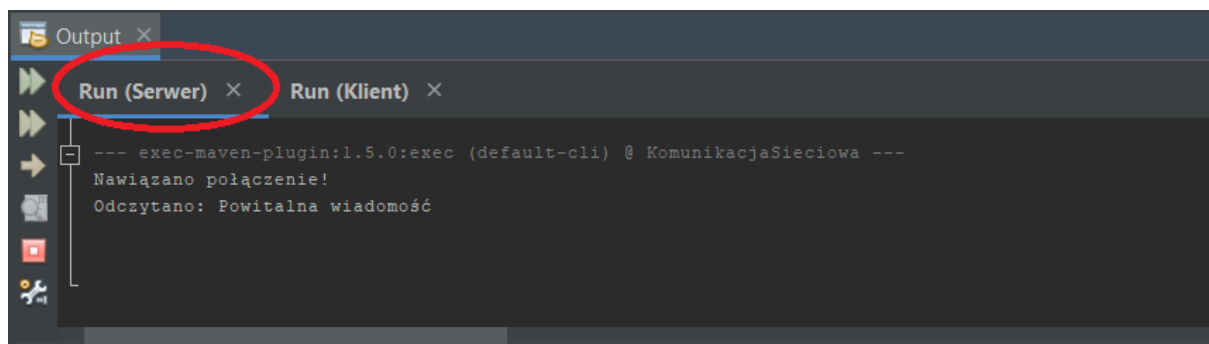
Aplikacja po uruchomieniu wygląda jak przedstawiono na rysunku 3.1. Użytkownik wprowadza tekst wiadomości w pole tekstowe. Po kliknięciu przycisku „Wyslij” wiadomość jest wysyłana na serwer. Wszystkie wiadomości znajdujące się na serwerze zostają pobrane i wyświetlone w wielowierszowym polu tekstowym. Na konsoli wyjściowej można zaobserwować przesłane wiadomości. Konsole wyjściowe poszczególnych części (klient, serwer) pokazano na rysunku 3.2 – 3.



Rys.3.1. Aplikacja wykorzystująca komunikację sieciową.



Rys.3.2. Konsola wyjściowa – klient.



Rys.3.3. Konsola wyjściowa – serwer.

Zadanie 3.1. Komunikator sieciowy – GUI

Wykorzystaj podany kod do stworzenia aplikacji GUI opartej na formatce JavaFX dostępnej w środowisku NetBeans lub IntelliJ IDEA. Rozbuduj aplikację o dodatkowe komponenty (np. etykiety) wyświetlające podstawowe dane klienta/ów. Przed podłączeniem klienta do serwera żądaj potwierdzenia (np. w konsoli). Wprowadź niezbędne modyfikacje.

Zadanie 3.2. Komunikator sieciowy - szyfr

Zmodyfikuj zadanie 3.1 w taki sposób, aby wiadomość przed wysłaniem na serwer była zaszyfrowana przy użyciu szyfru Vigenere’a. Deszyfrowanie ma być dostępne wyłącznie po stronie klienta – na serwerze wiadomość ma zostać przechowywana w postaci zaszyfrowanej.

LABORATORIUM 4. TWORZENIE APLIKACJI BAZODANOWYCH.

Cel laboratorium:

Praktyczne tworzenie aplikacji bazodanowej.

Zakres tematyczny zajęć:

- Podstawowe polecenia SQL.
- Konfiguracja projektu.
- Tworzenie aplikacji do obsługi prostej bazy danych.

Pytania kontrolne:

1. Co jest potrzebne do obsługi bazy danych w projekcie Java.
2. Jakie znasz polecenia SQL?
3. Czy dane w bazie mogą się powtarzać?
4. Czy znasz jakieś biblioteki/frameworku usprawniające pracę z bazami danych?

Baza danych

Baza danych (ang. database) jest zbiorem danych uporządkowanych zgodnie z określonymi regułami. W przypadku relacyjnych baz danych są to dane w postaci tabeli wraz z relacjami pomiędzy nimi. Przechowywanie dużej ilości danych (powiązanych ze sobą różnymi relacjami) np. w pliku tekstowym nie jest zalecanym podejściem. W tym laboratorium zostanie przedstawiony sposób wykorzystania relacyjnych bazy danych w języku Java, bez wchodzenia w specyfikę języka SQL (ang. Structured Query Language) używanego do tworzenia i modyfikowania baz danych.

Niezbędne podstawy SQL

Język SQL został opracowany w latach 70., a więc ponad 20 lat przed stworzeniem Javy 1.0. Od tego czasu był wprowadzany w wielu projektach komercyjnych i wiązało się to z ciągłymi modyfikacjami.

Podstawowymi poleceniami są:

CREATE – tworzy tabelę

DROP – usuwa tabelę

INSERT – umieszcza dane w tabeli

UPDATE – modyfikuje wiersze w tabeli

SELECT – pobiera dane z tabeli

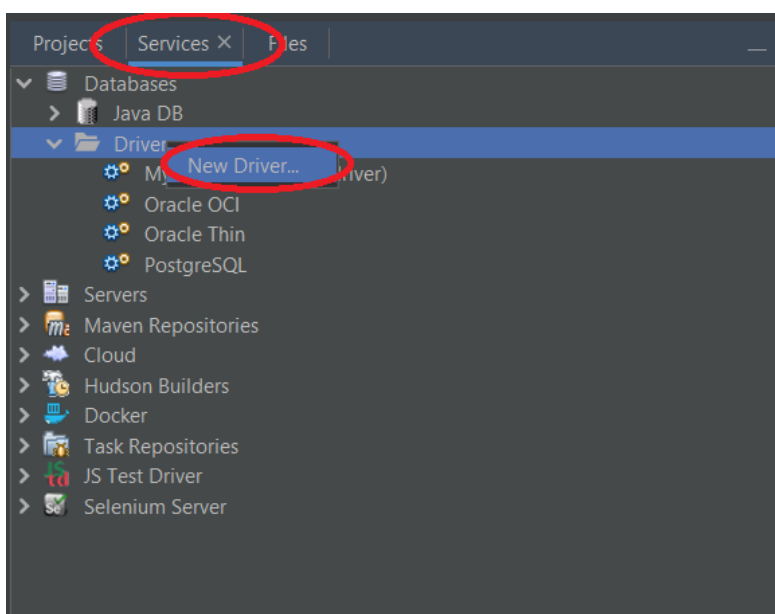
DELETE – usuwa rekord z tabeli

Naturalnie język SQL jest dużo bardziej rozbudowany ale na potrzeby utworzenia i modyfikacji prostej bazy danych przy użyciu oprogramowania napisanego w języku Java powyższe polecenia będą wystarczające.

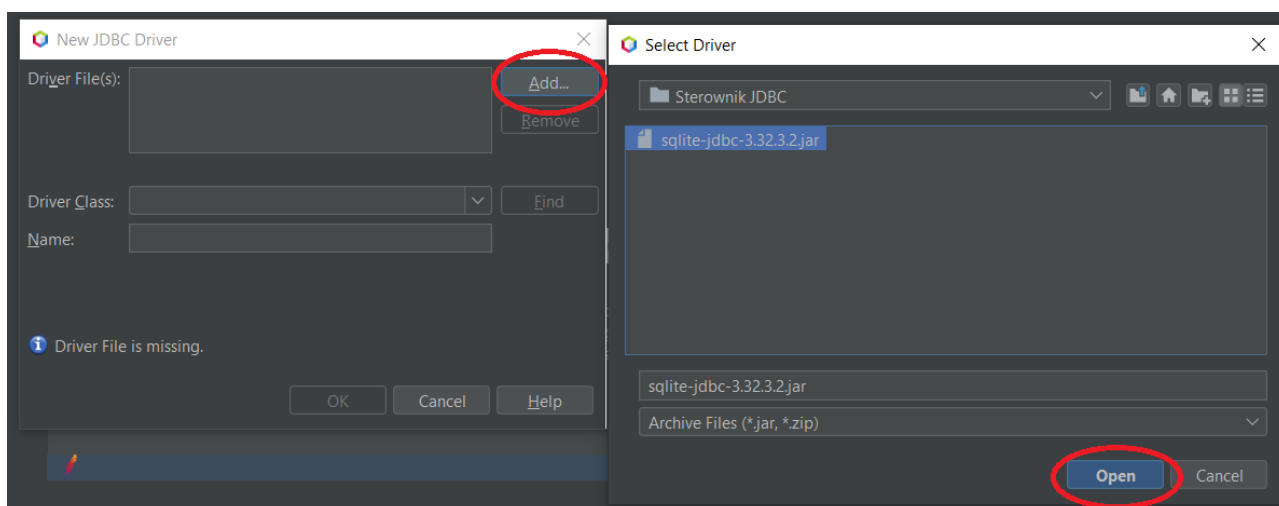


Konfiguracja projektu

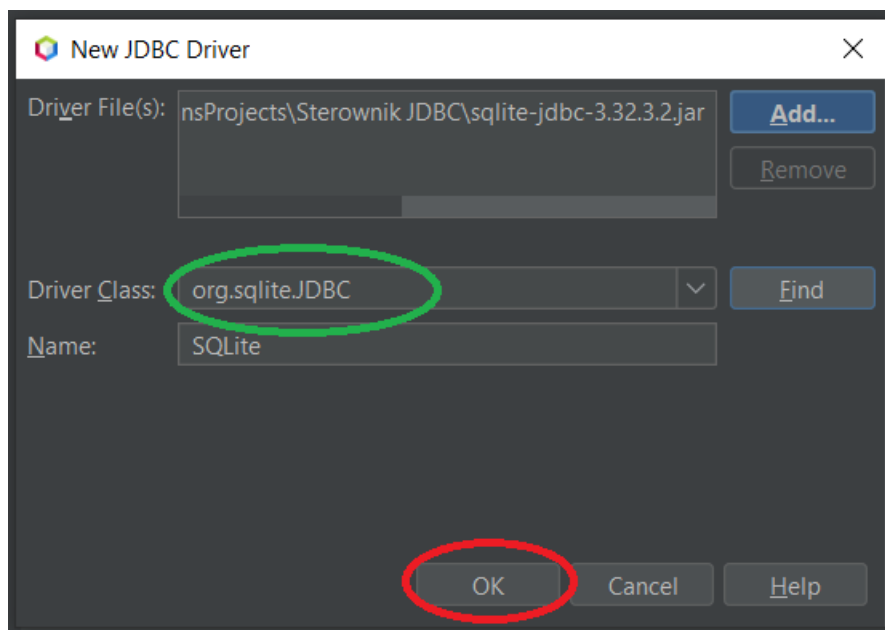
Korzystanie z bazy danych SQL w projekcie Java wymaga użycia specjalnej biblioteki ze sterownikiem JDBC (ang. Java DataBase Connectivity). W zależności od rodzaju bazy danych wymagany jest inny sterownik – dla przykładu zostanie wykorzystany sterownik do bazy danych SQLite. Jest to mała, szybka, wysoce wydajna i niezawodna baza danych (jest to baza bezserwerowa). Format pliku SQLite jest stabilny, wieloplatformowy i wstecznie kompatybilny. Sterownik JDBC można pobrać ze zdalnego [repozytorium](#) i dodać ręcznie do projektu lub wykorzystać funkcjonalność Maven'a do zarządzania zależnościami (wówczas należy dodać odpowiednią informację w pliku pom.xml). Dodanie ręczne pokazano na rysunku 4.1. – 3.



Rys.4.1. Dodanie sterownika JDBC – zakładka Services.



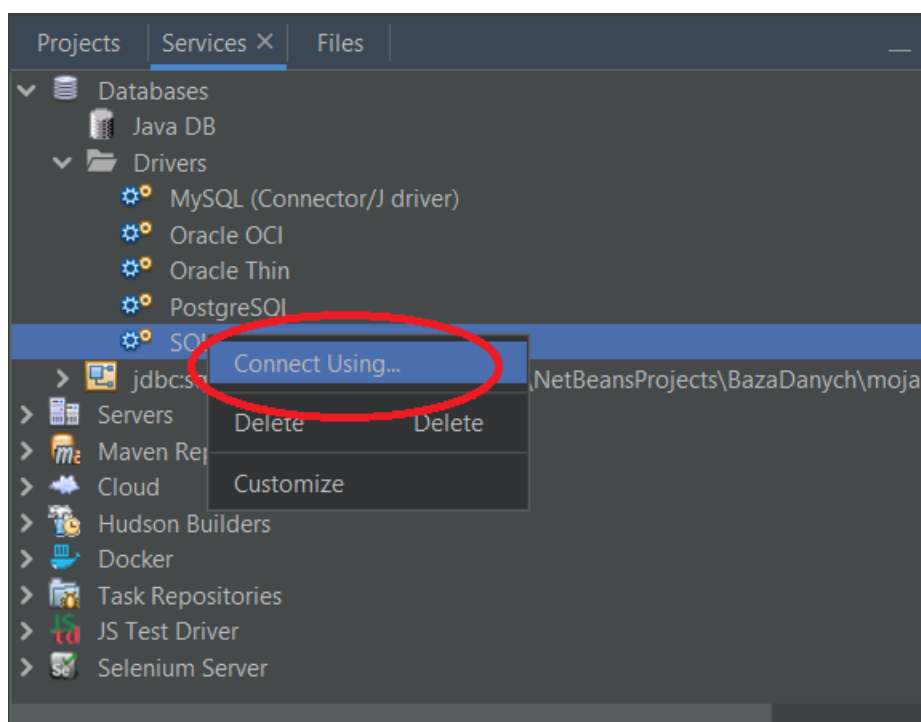
Rys. 4.2. Dodanie pobranego pliku z rozszerzeniem *.jar.



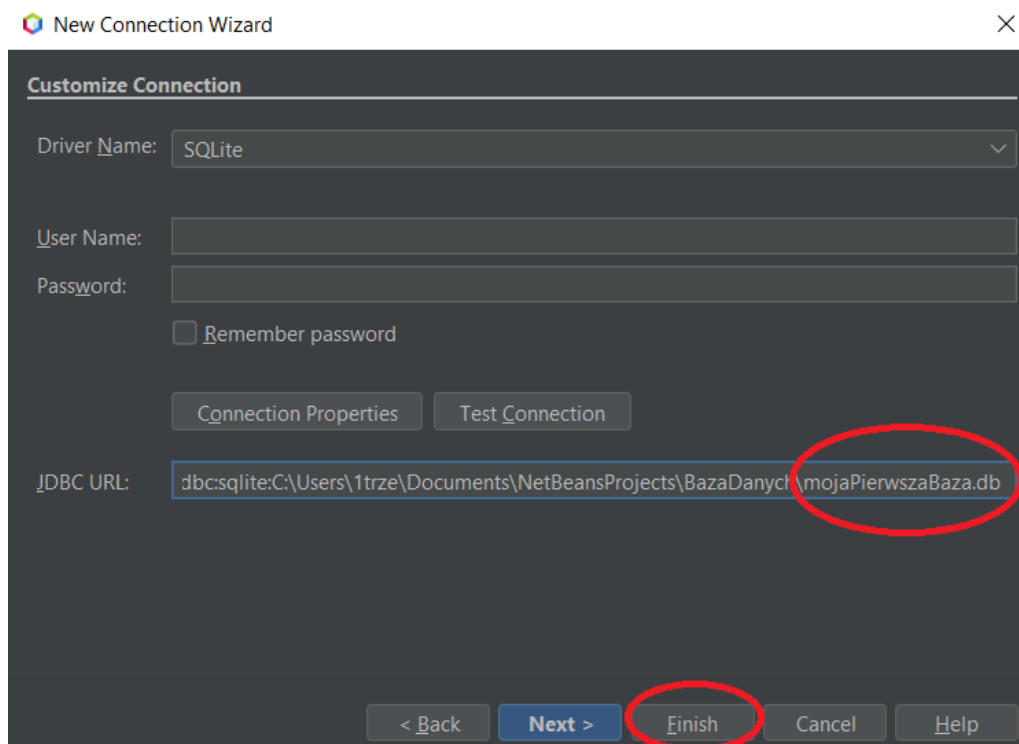
Rys.4.3. Dodany sterownik wraz z zaznaczoną na zielono nazwą klasy.

Po dodaniu sterownika należy utworzyć połączenie klikając PPM na dodany sterownik SQLite → Connect Using...(rys. 4.4), w kolejnym kroku (rys. 4.5) podając ścieżkę dostępu do bazy danych:

jdbc:sqlite:C:\Users\...\NetBeansProjects\BazaDanych\mojaPierwszaBaza.db

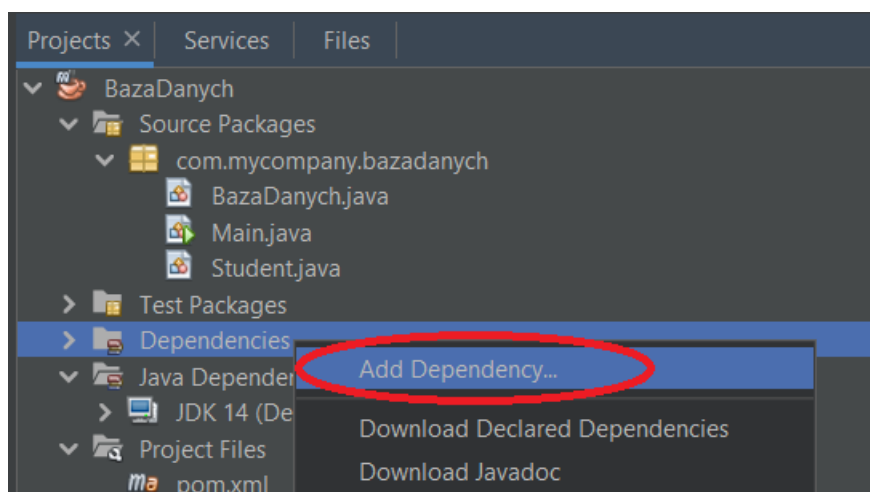


Rys.4.4. Tworzenie połączenia.

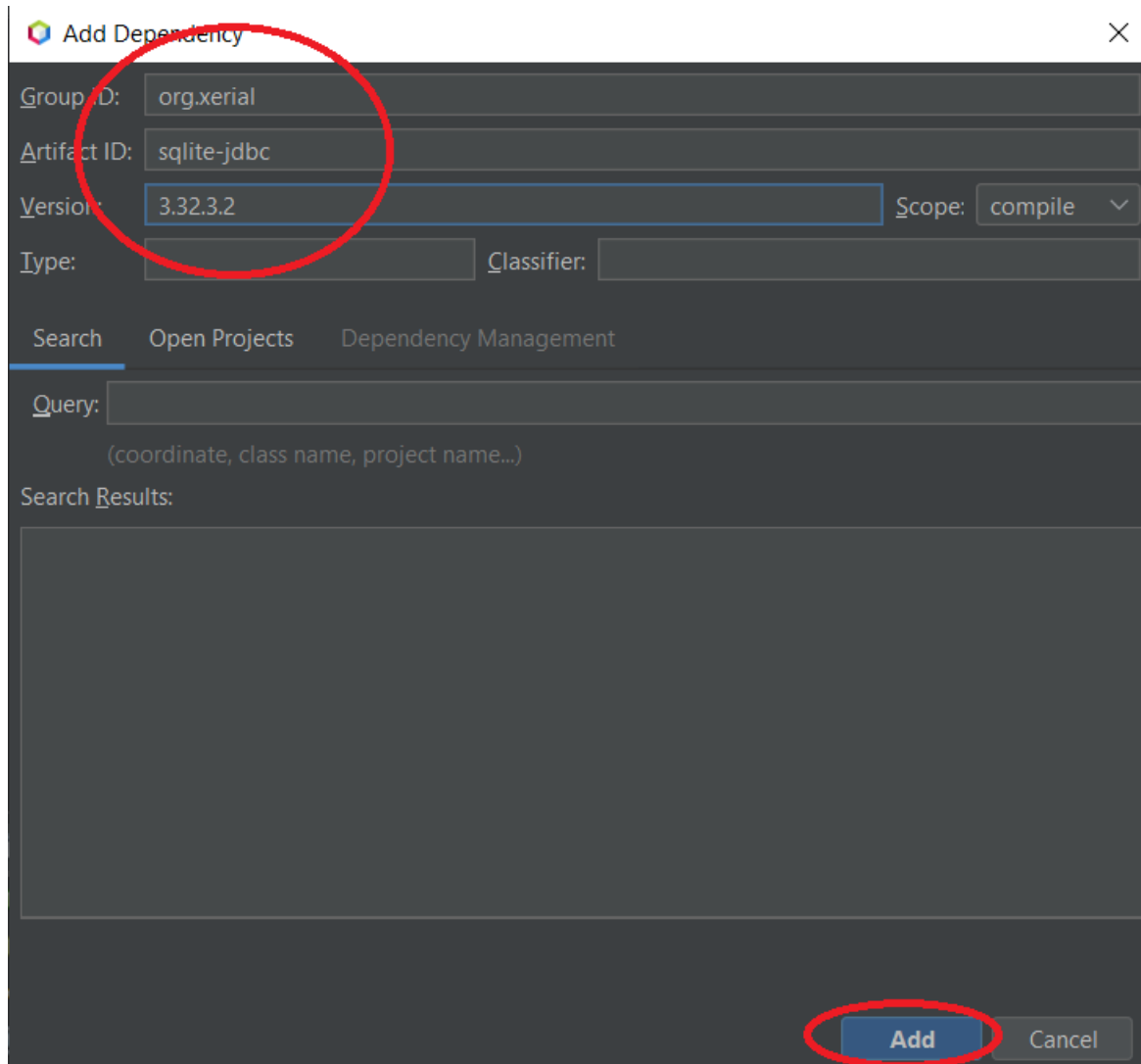


Rys.4.5. Dodanie ścieżki do bazy danych.

Alternatywą dla pobierania i ręcznego dodawania sterownika JDBC jest wykorzystanie Maven'a – w pliku pom.xml należy dodać zależność jak pokazano poniżej (rys. 4.6 – 7). Numer wersji sterownika można sprawdzić w [repozytorium Maven'a](#) – warto korzystać z najnowszej. Po dodaniu zależności sterownik zostanie pobrany.



Rys.4.6. Dodanie zależności – sterownika JDBC.



Rys.4.7. Podanie danych niezbędnych do dodania zależności.

Tworzenie bazy danych

Dla lepszego zapoznania się z tematyką wykorzystywania baz danych w aplikacjach Java poniżej przedstawiono przykład. Projekt składa się z klasy zarządzającej bazą danych (BazaDanych), klasy pomocniczej (Student) oraz klasy głównej (Main). W klasie zarządzającej zdefiniowano cztery metody:

- tworzenie tabeli STUDENCI (tworzTabele())
- wstawianie danych (wstawDane())
- pobieranie danych (pobierzDane())
- zamykanie połączenia (zamknijPolaczenie())

Korzystając z klasy BazaDanych oraz klasy Student możliwe jest utworzenie tabeli STUDENCI oraz dodawanie i odczyt danych z bazy. W metodzie main() zrealizowano przykładowe wstawianie i odczyt danych.

```
package com.mycompany.bazadanych;
```

```
import java.sql.*;
import java.util.LinkedList;
import java.util.List;

public class BazaDanych {

    private static final String DRIVER = "org.sqlite.JDBC";
    private static final String DB_URL =
"jdbc:sqlite:mojaPierwszaBaza.db";

    private Connection connection;
    private Statement statement;

    public BazaDanych() {
        try {
            Class.forName(DRIVER);
        } catch (ClassNotFoundException ex) {
            System.err.println("Brak sterownika JDBC");
            ex.printStackTrace();
        }
        try {
            connection = DriverManager.getConnection(DB_URL);
            statement = connection.createStatement();
            tworzTabele();
        } catch (SQLException ex) {
            System.err.println("Problem z otwarciem
połączenia");
            ex.printStackTrace();
        }
    }

    public boolean tworzTabele() {

        String tworz = "CREATE TABLE IF NOT EXISTS STUDENCI(id
INTEGER PRIMARY KEY AUTOINCREMENT, nazwisko String, imie
String)";
        try {
            statement.execute(tworz);
        } catch (SQLException e) {
            System.err.println("Błąd przy tworzeniu tabeli");
            e.printStackTrace();
            return false;
        }

        return true;
    }

    public boolean wstawDane(String tabela, String nazwisko,
String imie) {
```



```
        try {
            PreparedStatement preparedStatement =
connection.prepareStatement("INSERT INTO " + tabela + " VALUES
(null,?,?)");

            preparedStatement.setString(1, nazwisko);
            preparedStatement.setString(2, imie);

            preparedStatement.execute();

        } catch (SQLException e) {
            System.err.println("Błąd przy wprowadzaniu danych
studenta: " + nazwisko + " " + imie);
            e.printStackTrace();
            return false;
        }

        return true;
    }

    public List<Student> pobierzDane(String tabela) {

        List<Student> wyjscie = new LinkedList<Student>();
        try {
            ResultSet resultSet =
statement.executeQuery("SELECT * FROM " + tabela);
            int id;
            String nazwisko, imie;
            while (resultSet.next()) {
                id = resultSet.getInt("id");
                nazwisko = resultSet.getString("nazwisko");
                imie = resultSet.getString("imie");

                wyjscie.add(new Student(id, nazwisko, imie));
            }
        } catch (SQLException e) {
            System.err.println("Problem z wczytaniem danych z
BD");
            e.printStackTrace();
            return null;
        }
        return wyjscie;
    }

    public void zamknijPolaczenie() {
        try {
            connection.close();
        }
    }
}
```



```
        } catch (SQLException e) {  
            System.err.println("Problem z zamknięciem  
połączenia");  
            e.printStackTrace();  
        }  
    }  
}
```

```
package com.mycompany.bazadanych;  
  
public class Student {  
  
    private int id;  
    private String nazwisko, imie;  
  
    public Student(int id, String nazwisko, String imie) {  
        this.id = id;  
        this.nazwisko = nazwisko;  
        this.imie = imie;  
    }  
  
    public int getId() {  
        return id;  
    }  
  
    public void setId(int id) {  
        this.id = id;  
    }  
  
    public String getNazwisko() {  
        return nazwisko;  
    }  
  
    public void setNazwisko(String nazwisko) {  
        this.nazwisko = nazwisko;  
    }  
  
    public String getImie() {  
        return imie;  
    }  
  
    public void setImie(String imie) {  
        this.imie = imie;  
    }  
}
```



```
package com.mycompany.bazadanych;

import java.util.List;

public class Main {

    public static void main(String[] args) {

        BazaDanych bazaDanych = new BazaDanych();

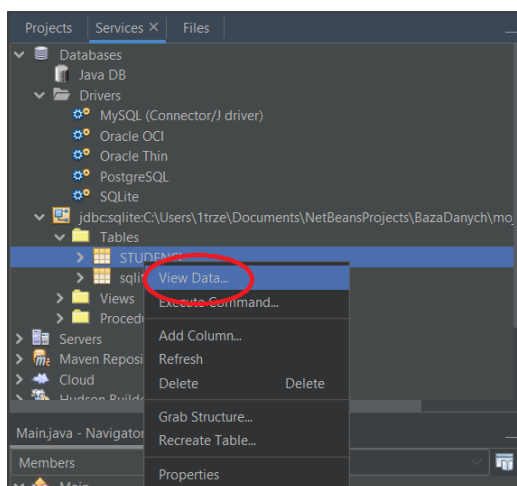
        bazaDanych.wstawDane("STUDENCI", "Kowalski", "Jan");
        bazaDanych.wstawDane("STUDENCI", "Wiśniewski",
"Piotr");
        bazaDanych.wstawDane("STUDENCI", "Nowak", "Michał");

        List<Student> lista =
bazaDanych.pobierzDane("STUDENCI");

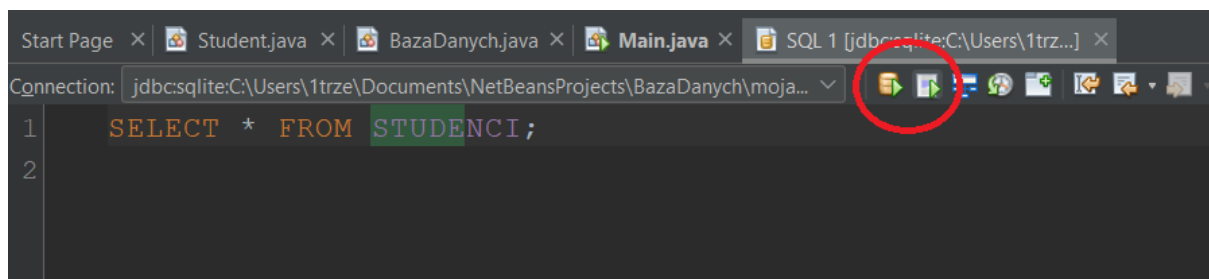
        for (Student s : lista) {
            System.out.println(s.getId() + " " +
s.getNazwisko() + " " + s.getImie());
        }
        bazaDanych.zamknijPolaczenie();
    }
}
```

Po uruchomieniu aplikacji zostanie utworzona instancja klasy BazaDanych, a następnie zostaną wywołane na niej metody: trzykrotnie wstawDane() oraz jednokrotnie pobierz dane. W pętli for – each pobrana lista zostanie wyświetlona.

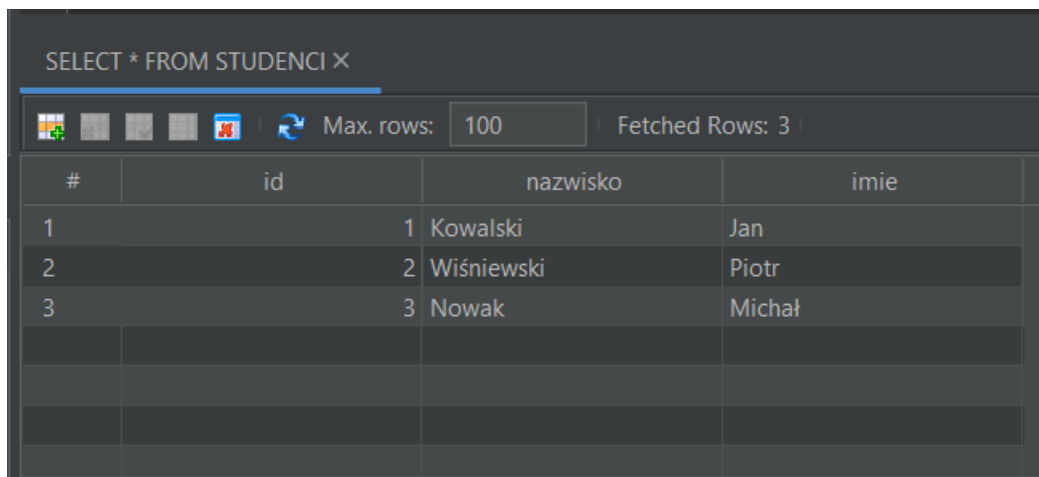
Środowisko NetBeans umożliwia podgląd danych znajdujących się w bazie – w zakładce Services (rys. 4.8) należy wybrać podłączoną bazę i na konkretnej tabeli kliknąć PPM → View Data, a następnie wybrać Run SQL lub Run Statement (rys.4.9). Otrzymane wyniki przedstawiono na rysunku 4.10.



Rys.4.8. Podgląd danych z tabeli.



Rys.4.9. Utworzenie zapytania SQL.



#	id	nazwisko	imie
1	1	Kowalski	Jan
2	2	Wiśniewski	Piotr
3	3	Nowak	Michał

Rys.4.10. Wynik zapytania.

Dane mogą być również widoczne z poziomu dowolnego programu obsługującego bazy danych (np. SQLite Studio).

Zadanie 4.1. Baza danych - rozwinięcie

Wykorzystując kod dostępny w instrukcji dokonaj niezbędnych modyfikacji. Zmieniony kod wykorzystaj do stworzenia aplikacji, która będzie mogła z poziomu konsoli:

- dodawać dane studentów do bazy
- odczytywać dane konkretnych studentów z bazy
- modyfikować dane studentów w bazie (w tym usuwać rekordy)

Zadbaj o obsługę wyjątków, upewnij się, że nie można dodawać danych pustych lub niepoprawnych (podstawowa walidacja), np. nie powinien powstać rekord z wartością w kolumnie imię studenta: Piotr2022. W przypadku pustej bazy danych wyświetl informacje o braku rekordów.

Zadanie 4.2. Baza danych - GUI

Wykorzystaj kod z zadania 13.1 do stworzenia aplikacji GUI. Dodaj niezbędne komponenty (JTextField, JButton, JTextArea...). Aplikacja ma posiadać funkcjonalności wymienione w zadaniu 1.1 oraz przyjemny dla oka graficzny interfejs użytkownika.

LABORATORIUM 5. INTERNACJONALIZACJA W JĘZYKU JAVA.

Cel laboratorium:

Zapoznanie z pojęciem internacjonalizacji w języku Java.

Zakres tematyczny zajęć:

- Internacjonalizacja.
- Aplikacja wielojęzyczna.

Pytania kontrolne:

1. Czym jest i18n?
2. W jaki sposób utworzyć plik z tłumaczeniem?
3. Jaką strukturę ma plik językowy?

Internacjonalizacja

Internacjonalizacja pozwala na stworzenie w ramach jednej aplikacji wielu wersji językowych. Jako ciekawostka – jest to tzw. rozwiązanie i18n (pomiędzy końcowymi literami w słowie internationalization jest 18 znaków).

W celu utworzenia wielu wersji językowych dla już gotowej aplikacji należy odpowiednio sprepować pliki o nazwie `messages_język.properties` np. `messages_pl.properties` i umieścić go w katalogu `resources/internationalization` lub `resources/i18n`. Wybrane środowiska wspomagają upakowanie wielu tłumaczeń w jedną ‘wiązkę’ tzw. Bundle. Przykładowy plik z tłumaczeniem (dla wygody alfabetycznie) został umieszczony poniżej:

```
action=\u00a7* {0} \u00a7{1}
addedToAccount=\u00a7{0} zostalo dodane do twojego konta.
addedToOthersAccount=\u00a7{0} zostalo dodane do konta
{1}\u00a7. Nowy stan konta\: {2}.
antiBuildInteract=\u00a7Nie masz uprawnień by oddziaływać z
{0}.
antiBuildUse=\u00a7Nie masz uprawnień by użyć {0}.
ignoredList=\u00a7Ignorowani\:\u00a7r {0}
illegalDate=Nieprawidłowy format daty.
infoUnknownChapter=\u00a7Nieznany rozdział.
insufficientFunds=\u00a7Nie posiadasz wystarczających
środków.
```

Wykorzystując bibliotekę do wstrzykiwania tzw. beanów możliwe jest przypisanie danemu obiektowi odpowiednich tłumaczeń.

```
public class Language {

    private Locale locale = new Locale("pl", "PL");
```



```
ResourceBundle resourceBundle =
ResourceBundle.getBundle("messages_pl.properties");

    public String getMessageDate(){
        return resourceBundle.getString("illegalDate");
    }
}
```

```
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;

class LanguageTest {

    @Test
    void getMessageDate() {
        Language language = new Language();

        Assertions.assertEquals(language.getMessageDate(), "Nieprawidlo
wy format daty.");
    }
}
```

Zadanie 5.1. Plik z tłumaczeniami

Dla przykładowej aplikacji (możesz wykorzystać autorską, która już istnieje) utwórz trzy pliki z tłumaczeniami – umieść je w odpowiednim katalogu.

Zadanie 5.2. Implementacja

Przeprowadź implementację tłumaczeń wykorzystując klasę ResourceBundle. Możesz posłużyć się klasą testową (laboratorium 12) lub zwykłą, która wyświetli dane tłumaczenia.

LABORATORIUM 6. TWORZENIE APLIKACJI Z WYKORZYSTANIEM BIBLIOTEKI LOMBOK.

Cel laboratorium:

Zapoznanie studentów z biblioteką Lombok. Tworzenie aplikacji z wykorzystaniem biblioteki Lombok.

Zakres tematyczny zajęć:

1. Wprowadzenie do biblioteki Lombok.
2. Wybrane adnotacje biblioteki Lombok.
3. Przypadki użycia.

Pytania kontrolne:

1. W jaki sposób dołączyć bibliotekę Lombok do projektu?
2. W jaki sposób tworzy się adnotacje? Podaj przykładowe adnotacje.
3. Do czego służą adnotacje `@Data`, `@NoArgsConstructor`, `@ToString`?
4. Czy adnotacje mogą być stosowane zamiennie?
5. Na jakim etapie – przed kompilacją, w trakcie kompilacji czy w trakcie użycia (runtime) adnotacje generują kod? Czy można to zmienić?
6. Czym charakteryzuje się wzorzec Builder?
7. Czy biblioteka Lombok może współpracować z innymi w jednym projekcie?

Wstęp

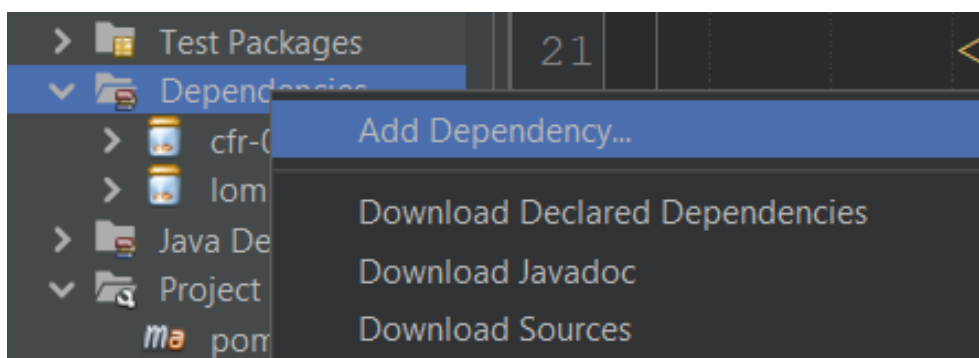
Język Java ma bardzo wiele zalet takich jak niezależność od platformy systemowej (i/lub dopasowanie do platformy sprzętowej poprzez odpowiednie biblioteki), duża wydajność, uniwersalność, współbieżność i wiele innych. Jednak czasami zdarza się, że do wykonania typowych zadań niezbędne jest napisanie wielu (jak się okazuje) zbędnych linii kodu. Takie podejście nie wnosi nic nowego do faktycznej wartości projektu, a jest jedynie ‘napełnianiem linii’. Do zwiększenia produktywności można wykorzystać bibliotekę Lombok. Odpowiada ona za automatyczne generowanie kodu bajtowego Java w plikach *.class po odpowiednim podłączeniu (uzależnieniu) projektu na podstawie tylko adnotacji. Dołączenie biblioteki wygląda bardzo podobnie jak w innych przypadkach. W projektach opartych na Maven’ie wystarczy dodać odpowiednią zależność (ang. dependency) uwzględniając nazwę i wersję biblioteki w pliku pom.xml. Poniższy kod może ułatwić zadanie:

```
<dependencies>
...
<dependency>
  <groupId>org.projectlombok</groupId>
  <artifactId>lombok</artifactId>
  <version>1.18.24</version>
  <scope>provided</scope>
</dependency>
...
</dependencies>
```

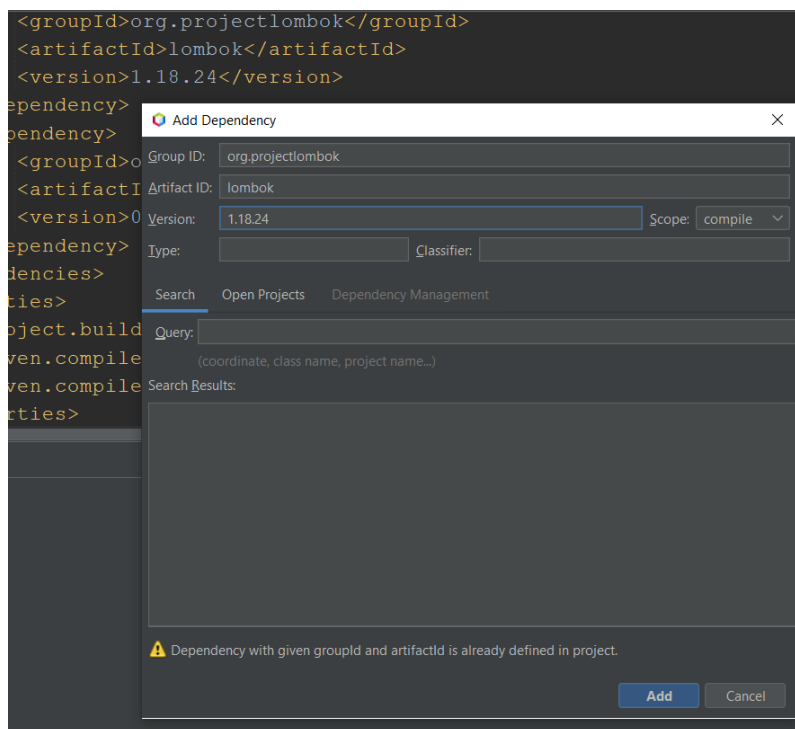


Więcej informacji można znaleźć na stronie [Projektu Lombok](#). Są tam szczegółowe instrukcje odnośnie instalacji biblioteki w różnych środowiskach bazując na pliku lombok.jar lub poleceniach Maven'a. Należy zauważyć, że dodając zależność od biblioteki Lombok nie uzależniamy jednocześnie wygenerowanych plików *.jar, więc użytkownicy nie będą zależni od tej biblioteki. Dzieje się tak dlatego, że jak zostało już wspomniane biblioteka jest używana na etapie kompilacji do generowania kodu na podstawie adnotacji. Nie jest to więc powiązanie ze środowiskiem wykonawczym.

Drugim sposobem jest dodanie zależności nie bezpośrednio w pliku pom.xml ale w projekcie, jak pokazano na rysunku 6.1 i 6.2. Jeżeli biblioteka jest już dołączona do projektu zostanie wyświetlony komunikat: „Dependency with given groupId and artifactId is already defined in project”. Konkurencyjne środowiska dla NetBeans, jak np. IntelliJ IDEA pozwalają na dołączenie tzw. pluginów, które skutecznie pomagają w pracy. Przykładem jest plugin Lombok, który pozwala wykorzystywać adnotacje Lombok'a przed wytworzeniem pliku *.class – jeszcze przed kompilacją.



Rys. 6.1. Dodawanie zależności w projekcie.



Rys. 6.2. Dodawanie biblioteki Lombok.

Enkapsulacja pól (właściwości) obiektów za pomocą publicznych metod pobierających i ustawiających (getter i setter) jest bardzo powszechną praktyką w świecie Javy. Dodatkowo wiele bibliotek w dużym stopniu korzysta z wzorca tzw. „Java Bean” (klasa z pustym konstruktorem i metodami pobierania/ustawiania dla pól). Większość środowisk IDE obsługuje automatyczne generowanie kodu dla tych wzorców (Alt+Insert →...) jednak ten kod musi po pierwsze zostać wygenerowany, po drugie musi znajdować się w naszych źródłach (dodatkowe linie kodu) i po trzecie musi być utrzymywany po dodaniu nowej właściwości lub zmianie nazwy pola (refaktoryzacja kodu bywa uciążliwa wraz z rozrostem klasy). Poniżej przedstawiono klasę bez wykorzystania biblioteki Lombok:

```
public class Person {

    private String name;
    private String surname;
    private int age;

    public Person() {
    }

    public Person(String name, String surname, int age) {
        this.name = name;
        this.surname = surname;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getSurname() {
        return surname;
    }

    public void setSurname(String surname) {
        this.surname = surname;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }
}
```

Przedstawiony przykład to 40 linii kodu, a zawiera tylko klasę Person składającą się z:



- 3 pól,
- 2 konstruktorów,
- zestawu getterów/seterów (w sumie 6)

Klasa ta jest dosyć prymitywna ale łatwo wyobrazić sobie, że może zostać rozwinięta do większej postaci, jednak jej znaczenie w modelu biznesowym nie zmieni się. Nawet po dodaniu kolejnych 3 pól ‘lepiej’ definiujących daną osobę będzie to tylko klasa opisowa do uzyskiwania/ustawiania informacji o osobie tj.: imię, nazwisko, wiek oraz dalszych informacji. Po dodaniu kolejnych pól na nowo muszą zostać zdefiniowane konstruktory, metody pobierające/ustawiając. I tak za każdym razem, kiedy klasa rozrośnie się o jedno lub dwa pola. Tak opisany rozwój klasy dużo łatwiej wyobrazić sobie wykorzystując bibliotekę Lombok. Odpowiednie adnotacje informują kompilator, że należy wygenerować odpowiednio metody typu get/set oraz konstruktor bez argumentów. Oczywiście istnieje odpowiednik adnotacji dla konstruktora z argumentami (nawet jeśli zmieni się liczba pól w klasie adnotacja pozostanie taka sama). Warto zauważyć na różnicę pomiędzy pokazanymi klasami, zawierającymi z punktu widzenia kompilatora to samo. Z punktu widzenia programisty pierwszy kod wymaga relatywnie dużego nakładu pracy przy wytworzeniu, a jeszcze więcej przy utrzymaniu w porównaniu z drugim.

```
@Getter @Setter @NoArgsConstructor
public class Person {

    private String name;
    private String surname;
    private int age;

    public Person(String name, String surname, int age) {
        this.name = name;
        this.surname = surname;
        this.age = age;
    }

}
```

Należy oczywiście pamiętać o zaimportowaniu odpowiednich adnotacji poprzez instrukcje (o ile środowisko nie wyręczy nas poprzez automatyczny import w trakcie pisania kodu):

```
package com.mycompany.lombokproject;

import lombok.Getter;
import lombok.Setter;
import lombok.NoArgsConstructor;
```

Lub w wersji prostszej, jeśli nie zamierzamy tworzyć konfliktu nazw:

```
package com.mycompany.lombokproject;
```

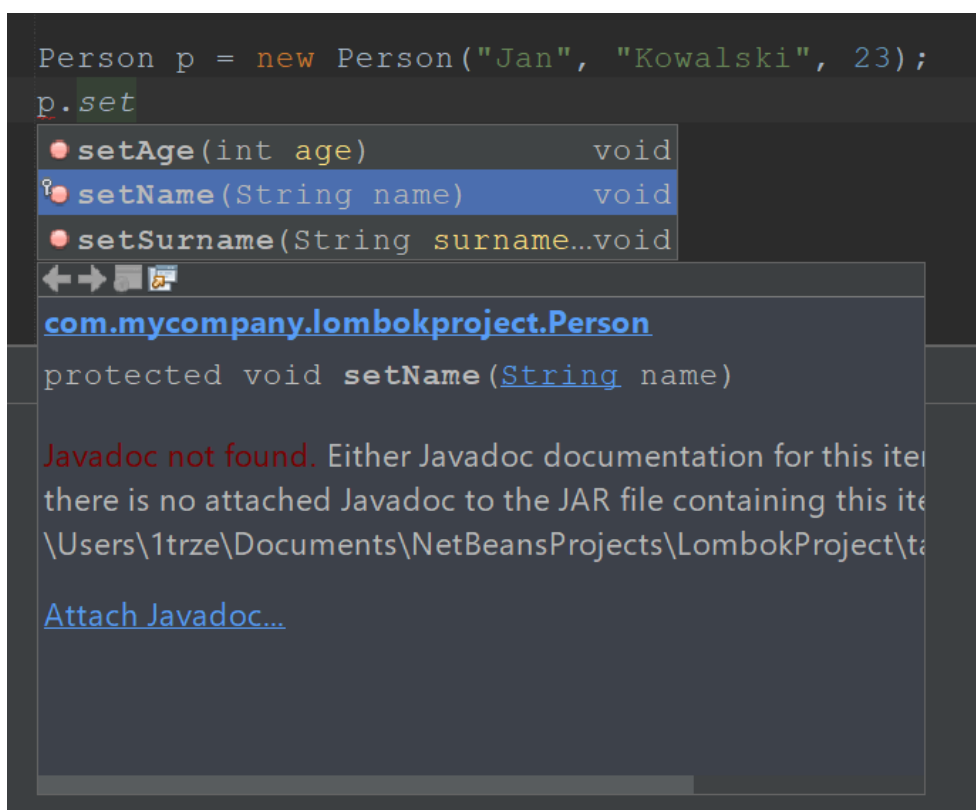


```
import lombok.*;
```

Warto dodać, że wygenerowane metody będą dostępne dopiero po pierwszej kompilacji po dodaniu adnotacji (wcześniej metoda np. `getName()` z klasy `Person` nie będzie istniała) – chyba, że projekt jest tworzony w środowisku IntelliJ IDEA, gdzie od razu po dodaniu adnotacji metody te są widoczne (dzięki zainstalowanemu plugin'owi Lombok).

Próba specyficznego ustawienia widoczności dla niektórych pól jest również możliwa. Przykładowo ustawiając modyfikatory dostępu do pola `name` jako chronione (`protected`), bo takie są wymagania wystarczy że zastosujemy bardziej specyficzną adnotację:

```
private @Setter(AccessLevel.PROTECTED) String name;
```



Rys. 6.3. Setter pola `name` jako chroniony.

Wyraźnie widać na rysunku, że pozostałe metody ustawiające są dostępne w swojej standardowej wersji jako publiczne, podczas gdy `setName()` ma modyfikator `protected` zgodnie z wymaganiami.

Leniwe pobieranie

Kolejny przykład zastosowania biblioteki Lombok może zostać przedstawiony za pomocą poniższej klasy. Operacje pobierania i zapisywania danych np. z pliku tekstowego lub bazy danych są stosunkowo kosztowne dlatego nie robi się tego nadmiarowo. Można zastosować podejście polegające na jednorazowym pobraniu danych, a następnie buforowaniu ich aby odczyt był możliwie szybki dzięki przechowywaniu ich w pamięci aplikacji. Inne podejście polega na pobraniu danych tylko wtedy, gdy są one niezbędne (po raz pierwszy). Ta

cecha nosi bardzo charakterystyczną nazwę – leniwość. Stosując leniwe pobieranie dane są zdobywane tylko przy pierwszym wywołaniu gettera.

Wybierając podejście pierwsze możemy dane przechowywać jako pola przygotowanej wcześniej klasy. Klasa ma za zadanie zapewnić dostęp do pola, dzięki czemu będziemy mogli zwracać dane z pamięci podręcznej. Jednak pobieranie wszystkich możliwych danych i wypełnianie nimi pól klasy może okazać się zbyt pracochłonne/kosztowne. Bazując na podejściu leniwym można wyobrazić sobie metodę pobierającą dane tylko wtedy, gdy pole nie jest jeszcze ‘uzupełnione’ czyli ma wartość null. Jest to tzw. lazy getter (leniwy getter). Stosując odpowiednią adnotację dzięki dobrodziejstwom Lomboka możemy w bardzo prosty sposób zaimplementować takie rozwiązanie.

```
public class GetterLazy {

    @Getter(lazy = true)
    private final Map<String, Long> transactions =
getTransactions();

    private Map<String, Long> getTransactions() {

        final Map<String, Long> cache = new HashMap<>();
        List<String> txnRows = readTxnListFromFile();

        txnRows.forEach(s -> {
            String[] txnIdValueTuple = s.split(" ");
            cache.put(txnIdValueTuple[0],
Long.parseLong(txnIdValueTuple[1]));
        });

        return cache;
    }

    private List<String> readTxnListFromFile() {
        /*
        ...
        */
        return list;
    }
}
```

Po kompilacji kod może wyglądać następująco:

```
public class GetterLazy {

    private final AtomicReference<Object> transactions = new
AtomicReference();

    public GetterLazy() {
    }
}
```



```
//other methods

public Map<String, Long> getTransactions() {
    Object value = this.transactions.get();
    if (value == null) {
        synchronized(this.transactions) {
            value = this.transactions.get();
            if (value == null) {
                Map<String, Long> actualValue =
this.readTxnsFromFile();
                value = actualValue == null ?
this.transactions : actualValue;
                this.transactions.set(value);
            }
        }
    }

    return (Map)((Map)(value == this.transactions ? null :
value));
}

private Map<String, Integer> readTxnsFromFile() {

    /*
    ...
    */
    return outMap;
}

}
```

Takie rozwiązanie powoduje, że dostęp do danych jest możliwy za pomocą gettera ale tylko w przypadku, gdy ta wartość jest pusta (null). Jeżeli dane w pliku nie ulegną zmianie nie będzie sytuacji, w której wciąż aktualizowane są pola tymi samymi danymi. Warto zauważyć, że biblioteka Lombok opakowuje pole w obiekt klasy AtomicReference. Dzięki temu aktualizacja pola jest niepodzielna. Metoda getNames() zapewnia odczytanie danych, jeśli mają one wartość null. Nie jest polecanym rozwiązaniem używanie pola AtomicReference z poziomu klasy – lepiej jest uzyskać dostęp do pola z użyciem metody np. getNames(). Stosując inne adnotacje takie jak np. @ToString w tej samej klasie metoda getNames() może zostać użyta, zamiast zapewnienia bezpośredniego dostępu do pola.

Data Transfer Object

Kolejnym często spotykanym typem klasy opisującym obiekty jest tzw. DTO – Data Transfer Object. Klasa ta charakteryzuje się posiadaniem wyłącznie pól (opisujących dane) oraz metod pobierających (zestaw getterów). Brak jest tu jakiegokolwiek logiki biznesowej. Tak zaprojektowana klasa dostarcza strukturę danych, którą budujemy raz i nie zmieniamy jej nigdy. Przykładem może być klasa reprezentująca pomyślną operację logowania. Żadne pole

nie może mieć wartości null, a obiekty nie mogą być zmienne (stąd final). Poniżej zaprezentowano kod klasy w wersji „tradycyjnej” oraz przy wykorzystaniu biblioteki Lombok.

```
import java.net.URL;
import java.time.Duration;
import java.time.Instant;
import java.util.Objects;

public class LoginResult {
    private final Instant loginTs;

    private final String authToken;
    private final Duration tokenValidity;

    private final URL tokenRefreshUrl;

    public LoginResult(Instant loginTs, String authToken,
Duration tokenValidity, URL tokenRefreshUrl) {
        this.loginTs = Objects.requireNonNull(loginTs, "not
null!");
        this.authToken = Objects.requireNonNull(authToken, "not
null!");
        this.tokenValidity =
Objects.requireNonNull(tokenValidity, "not null!");
        this.tokenRefreshUrl =
Objects.requireNonNull(tokenRefreshUrl, "not null!");
    }

    public Instant getLoginTs() {
        return loginTs;
    }

    public String getAuthToken() {
        return authToken;
    }

    public Duration getTokenValidity() {
        return tokenValidity;
    }

    public URL getTokenRefreshUrl() {
        return tokenRefreshUrl;
    }
}
```

Ta sama klasa z wykorzystaniem biblioteki Lombok:



Fundusze Europejskie
Wiedza Edukacja Rozwój



**Rzeczpospolita
Polska**

Unia Europejska
Europejski Fundusz Społeczny




```
import lombok.Getter;
import lombok.NonNull;
import lombok.RequiredArgsConstructor;
import lombok.experimental.Accessors;

import java.net.URL;
import java.time.Duration;
import java.time.Instant;

@RequiredArgsConstructor
@Accessors(fluent = true) @Getter
public class LoginResult {

    private final @NonNull Instant loginTs;

    private final @NonNull String authToken;
    private final @NonNull Duration tokenValidity;

    private final @NonNull URL tokenRefreshUrl;

}
```

Jak widać w wersji pierwszej kod zajmuje 38 wierszy, natomiast w wersji drugiej tylko 12 – dla kompilatora jest to to samo, a zajmuje ponad trzykrotnie mniej, przez co jest czytelniejszy oraz łatwiejszy w zarządzaniu. Adnotacja `@RequiredArgsConstructor` tworzy konstruktor dla wszystkich pól oznaczonych w klasie jako `final`. Z kolei adnotacja `@NonNull` spowoduje sprawdzenie ich pod kątem bycia nullem i odpowiednio wyrzuci `NullPointerException`. Warto zauważyć, że po dodaniu adnotacji `@Setter` oraz braku modyfikatora `final` również zostałby wyrzucony wyjątek `NullPointerException` (!). Adnotacja `@Accessors(fluent = true)` powoduje, że gettery mają taką samą nazwę jak nazwa pola, przykładowo: pole – `authToken`, getter – `authToken()` (zamiast `getAuthToken()`).

Builder

Często spotykanym wzorcem projektowym jest Builder. Zapewnia wieloetapowe budowanie obiektu, które następuje w efekcie pracy specjalnego obiektu budującego (buildera). Przykładem zastosowania tego wzorca może być klasa odpowiadająca za konfigurację klienta REST API.

```
public class ApiClientConfiguration {

    private String host;
    private int port;
    private boolean useHttps;

    private long connectTimeout;
    private long readTimeout;

    private String username;

}
```



```
private String password;

public ApiClientConfiguration() {
}

public ApiClientConfiguration(String host, int port,
boolean useHttps, long connectTimeout, long readTimeout,
String username, String password) {
    this.host = host;
    this.port = port;
    this.useHttps = useHttps;
    this.connectTimeout = connectTimeout;
    this.readTimeout = readTimeout;
    this.username = username;
    this.password = password;
}

public String getHost() {
    return host;
}

public void setHost(String host) {
    this.host = host;
}

public int getPort() {
    return port;
}

public void setPort(int port) {
    this.port = port;
}

public boolean isUseHttps() {
    return useHttps;
}

public void setUseHttps(boolean useHttps) {
    this.useHttps = useHttps;
}

public long getConnectTimeout() {
    return connectTimeout;
}

public void setConnectTimeout(long connectTimeout) {
    this.connectTimeout = connectTimeout;
}
```



```
public long getReadTimeout() {
    return readTimeout;
}

public void setReadTimeout(long readTimeout) {
    this.readTimeout = readTimeout;
}

public String getUsername() {
    return username;
}

public void setUsername(String username) {
    this.username = username;
}

public String getPassword() {
    return password;
}

public void setPassword(String password) {
    this.password = password;
}
}
```

Rezygnacja z zestawu setterów jest warta uwagi, bo po pierwsze spowodowałoby to mniej linii kodu (o całe 26 linii, w sumie z 84), a po drugie (ważniejsze) po zbudowaniu instancja nie miałaby możliwości być ponownie ustawioną, co czyniłoby ją niezmienną. Opcją numer dwa jest oczywiście wykorzystanie biblioteki Lombok dodając adnotację `@Builder`. Dzięki temu narzędziu konstruktor i wszystkie powiązane z nim metody ustawiające zostaną wygenerowane. Daje to całe 14 linii, a użycie zostało pokazane poniżej definicji klasy.

```
import lombok.Builder;

@Builder
public class ApiClientConfiguration {
    private String host;
    private int port;
    private boolean useHttps;

    private long connectTimeout;
    private long readTimeout;

    private String username;
    private String password;
}
```



```
*/
...
*/

ApiClientConfiguration config =
    ApiClientConfiguration.builder()
        .host("api.server.com")
        .port(443)
        .useHttps(true)
        .connectTimeout(15_000L)
        .readTimeout(5_000L)
        .username("myusername")
        .password("secret")
        .build();

/*
...
*/
```

Bardzo wygodnym rozwiązaniem jest użycie adnotacji `@Data` w przypadku klasy do transferu danych. Jest to substytut zbioru adnotacji jak np.: `@ToString`, `@Getter`, `@Setter`, `@EqualsAndHashCode` oraz `@RequiredArgsConstructor`.

Zadanie 6.1. Klasa modelowa – bez użycia biblioteki Lombok

Napisz klasę modelową – np. `Student`, `Pracownik`, `Osoba`. Klasa ma zawierać 4 pola prywatne, zestaw getterów i setterów oraz konstruktory bez parametrów i z wszystkimi parametrami. Następnie w klasie `Main`, w metodzie `main` utwórz kilka obiektów tej klasy. Po utworzeniu obiektów zmień właściwości w klasie modelowej – zmień liczbę pól klasy, zaktualizuj odpowiednio metody i konstruktory. Wykonaj jeszcze jedną zmianę liczby pól.

Zadanie 6.2. Klasa modelowa – z biblioteką Lombok

Wykonaj te same czynności co w zadaniu 6.1 ale zastosuj bibliotekę `Lombok` oraz niezbędne adnotacje. Czy po zmianie liczby pól w klasie modelowej należy coś zmieniać?

Zadanie 6.3. Dane od użytkownika

Napisz klasę wyposażoną w zestaw metod do pobierania danych od użytkownika – np. może to być prototyp logowania w wersji konsolowej lub dowolny inny ‘formularz’ (ewentualnie gra w zgadywanie liczby). Upewnij się, że dane pobierane przez metody nie są `null`’ami.

LABORATORIUM 7. TWORZENIE APLIKACJI Z WYKORZYSTANIEM BIBLIOTEKI JAVA FX.

Cel laboratorium:

Zapoznanie studentów z biblioteką JavaFX. Tworzenie aplikacji z wykorzystaniem biblioteki JavaFX.

Zakres tematyczny zajęć:

- Opis biblioteki JavaFX.
- Tworzenie projektu GUI.

Pytania kontrolne:

1. Jakie są podstawowe różnice między bibliotekami Swing oraz JavaFx?
2. W jaki sposób można utworzyć aplikację GUI?
3. Czy możliwe jest tworzenie własnych komponentów przy pomocy biblioteki JavaFX?
4. Co mają wspólnego kaskadowe arkusze stylów (CSS) z biblioteką JavaFX?

JavaFX

JavaFX jest biblioteką narzędzi GUI (Graphical User Interface – Graficzny Interfejs Użytkownika), która ułatwia tworzenie aplikacji wyposażonych w GUI. Jest to następca znanej biblioteki Swing, która jest starsza, nie wspiera wielu udogodnień (przykładowo grafiki wektorowej, CSS, definiowanie widoku aplikacji w języku XML).

JavaFX zawiera solidny zestaw komponentów GUI (przyciski, pola tekstowe, tabele, menu, wykresy i wiele innych). Dzięki definiowaniu widoku aplikacji przy pomocy XML'a można zapomnieć o konieczności określania rozmieszczenia elementów w kodzie plików *.java. Dodatkowo rozdzielenie widoku od logiki aplikacji pozwala na łatwiejszy podział obowiązków w zespole. Tworzenie elementów w oparciu o kaskadowe arkusze stylów jest wygodną opcją, metody typu setVisible(), setColor() nie są już niezbędne.

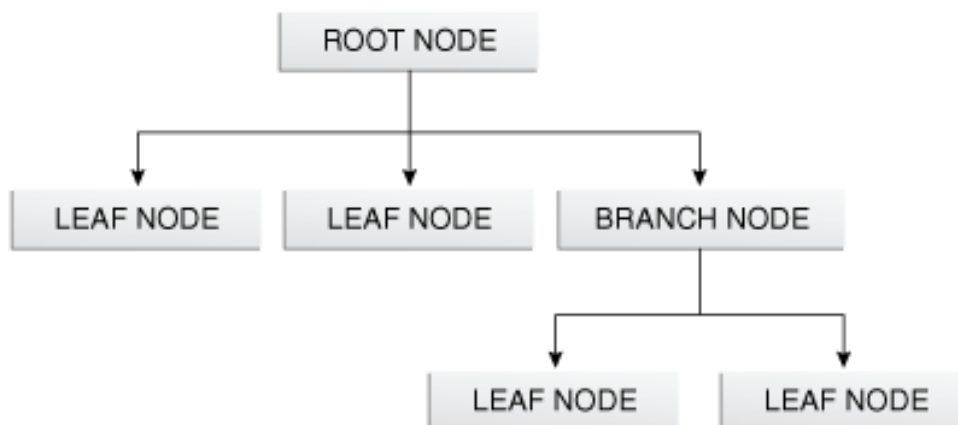
Narzędzia

Do tworzenia aplikacji przy użyciu JavaFX niezbędny jest Scene Builder. W przeszłości można było pobrać oprogramowanie bezpośrednio ze strony firmy Oracle, jednak obecnie możliwe jest to na stronie <https://gluonhq.com/products/scene-builder/>. W środowisku należy podać odpowiednio ścieżkę do zainstalowanej aplikacji. Dzięki temu możliwe jest otwieranie widoków za pomocą IDE, ale również uruchamianie Scene Buildera w oddzielnym oknie przez IDE.

Interfejs JavaFX Scene Graph API w dużym stopniu ułatwia tworzenie aplikacji wyposażonych w graficzny interfejs użytkownika, zwłaszcza jeśli chodzi o złożone efekty wizualne. Zamiast bezpośredniego wywoływania prymitywnych metod rysowania wykorzystuje się interfejs API do automatycznego obsługiwanie operacji renderowania. Dzięki temu zmniejsza się ilość kodu w aplikacji niezbędnego do jej poprawnego działania.

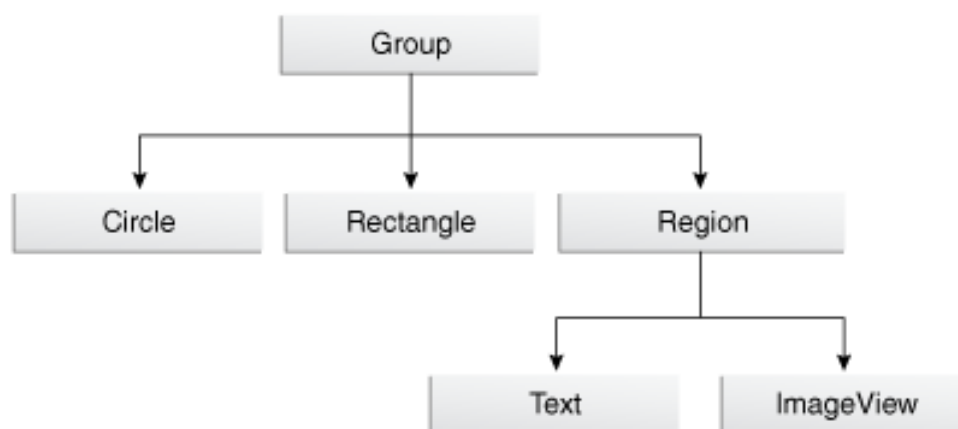


Poszczególne elementy znajdujące się na wykresie sceny (ang. Scene) są znane jako węzły. Każdy węzeł jest klasyfikowany albo jako węzeł gałęzi(ang. branch node), co oznacza że może mieć dzieci albo jako węzeł liścia (ang. leaf node), co oznacza, że nie może mieć dzieci. Pierwszy węzeł w drzewie jest nazywany zawsze węzłem głównym (ang. root node) i nigdy nie ma rodzica. Szczegółowy rozkład został przedstawiony na rysunku 7.1.



Rys. 7.1. Elementy sceny w JavaFX (źródło: <https://docs.oracle.com/javafx/2/scenegraph/jfxpub-scenegraph.htm>).

W API JavaFX zdefiniowana jest pewna liczba klas mogących pełnić rolę węzłów głównych, gałęzi lub liści. Po zastąpieniu schematu ogólnego faktycznie istniejącymi nazwami klas powyższy diagram może wyglądać jak na rysunku 7.2.



Rys. 7.2. Klasy jako elementy sceny w JavaFX (źródło: <https://docs.oracle.com/javafx/2/scenegraph/jfxpub-scenegraph.htm>).

Węzłem głównym będzie obiekt klasy Group, węzły bez dzieci (węzły liście) będą należały do klasy Circle oraz Rectangle. Obiekty klasy Region będą węzłem z dziećmi (węzeł gałąź) rozwidlającym się na dwie dodatkowe klasy Text oraz ImageView. Oczywiście to tylko przykładowy diagram i może on być dużo bardziej rozbudowany, ale podstawowa organizacja zawarta na nim jest niezbywalna – należy traktować to jako wzorzec, który będzie obecny we wszystkich aplikacjach.

Przykładowy kod umieszczony poniżej powoduje utworzenie okna przedstawionego na rysunku 7.3.

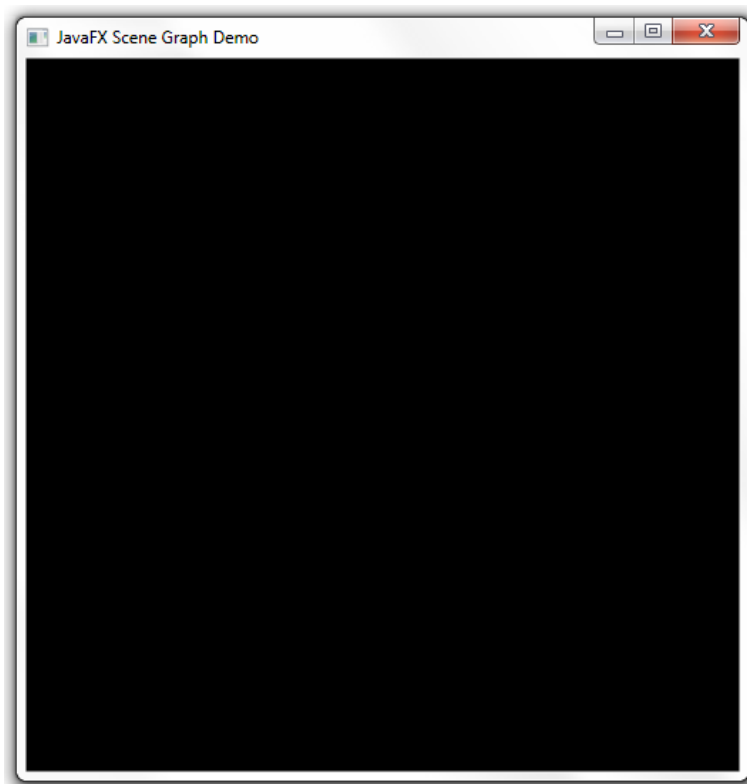
```
package scenegraphdemo;

import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.stage.Stage;

public class Main extends Application {

    @Override
    public void start(Stage stage) {
        Group root = new Group();
        Scene scene = new Scene(root, 500, 500, Color.BLACK);
        stage.setTitle("JavaFX Scene Graph Demo");
        stage.setScene(scene);
        stage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```



Rys. 7.3. Okno bez zawartości – pusta scena.



Istotne jest tutaj zwrócenie uwagi na kilka elementów:

- klasa Main jest rozszerzona o klasę Application (dziedziczy po niej), a zatem metoda start jest nadpisywana i jako parametr otrzymuje jedyny obiekt klasy Stage (najwyższy poziomem kontener GUI,
- węzeł główny (root node) jest tworzony i przekazywany do konstruktora sceny razem z parametrami okna (szerokość, wysokość, kolor wypełnienia),
- na obiekcie klasy Stage wywoływane są metody ustawiające tytuł, zawartość oraz widoczność,
- do wywołania głównej metody używana jest metoda launch().

Wynikiem działania tego kodu jest bardzo prosta aplikacja, która służy jedynie do prostej prezentacji w jaki sposób można uruchomić podstawowe okno. Węzeł główny nie ma w tym momencie dzieci, więc wewnątrz okna nic się nie znajduje. W celu rozbudowania aplikacji można dodać kod przedstawiony poniżej:

```
package scenegraphdemo;

import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.stage.Stage;

public class Main extends Application {

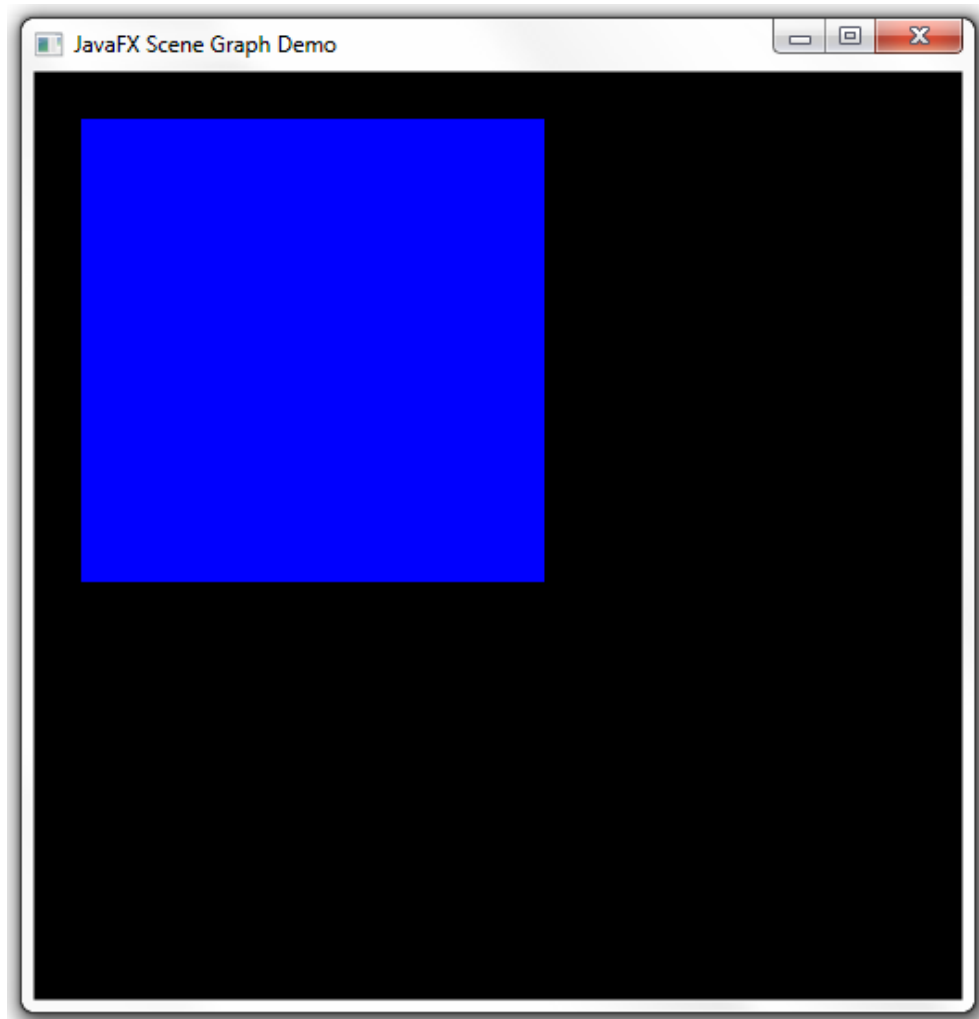
    @Override
    public void start(Stage stage) {
        Group root = new Group();
        Scene scene = new Scene(root, 500, 500, Color.BLACK);

        Rectangle r = new Rectangle(25,25,250,250);
        r.setFill(Color.BLUE);
        root.getChildren().add(r);

        stage.setTitle("JavaFX Scene Graph Demo");
        stage.setScene(scene);
        stage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```





Rys. 7.4. Okno z zawartością.

Przy stosunkowo niewielkim nakładzie kodu można uzyskać ciekawe efekty wizualne. Dzieje się tak dzięki zarządzaniu obiektami graficznymi przez wspomniany wcześniej diagram sceny, a więc dzięki hierarchii klas. Zanim sprawdzisz działanie poniższego kodu postaraj się rozszyfrować co powinno się stać z aplikacją.

```
package scenegraphdemo;

import javafx.animation.FillTransition;
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.stage.Stage;
import javafx.animation.Timeline;
import javafx.animation.ParallelTransition;
import javafx.animation.RotateTransition;
import javafx.animation.ScaleTransition;
```

```
import javafx.animation.TranslateTransition;
import javafx.util.Duration;

public class Main extends Application {

    @Override
    public void start(Stage stage) {
        Group root = new Group();
        Scene scene = new Scene(root, 500, 500, Color.BLACK);
        Rectangle r = new Rectangle(0, 0, 250, 250);
        r.setFill(Color.BLUE);
        root.getChildren().add(r);

        TranslateTransition translate =
            new TranslateTransition(Duration.millis(750));
        translate.setToX(390);
        translate.setToY(390);

        FillTransition fill = new
        FillTransition(Duration.millis(750));
        fill.setToValue(Color.RED);

        RotateTransition rotate = new
        RotateTransition(Duration.millis(750));
        rotate.setToAngle(360);

        ScaleTransition scale = new
        ScaleTransition(Duration.millis(750));
        scale.setToX(0.1);
        scale.setToY(0.1);

        ParallelTransition transition = new
        ParallelTransition(r,
            translate, fill, rotate, scale);
        transition.setCycleCount(Timeline.INDEFINITE);
        transition.setAutoReverse(true);
        transition.play();

        stage.setTitle("JavaFX Scene Graph Demo");
        stage.setScene(scene);
        stage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

Przedstawiony przykład można uznać za prymitywny, jednak zostały w nim poruszone kluczowe pojęcia – pakiet `javafx.scene` definiuje mnóstwo klas ale najistotniejsze to:

- węzeł (Node) – klasa bazowa dla wszystkich węzłów diagramu,
- rodzic (Parent) – klasa bazowa dla wszystkich węzłów gałęzi (bezpośrednio rozszerza Node),
- scena (Scene) – klasa kontener dla całej zawartości na diagramie sceny.

Wspomniane klasy definiują funkcjonalności, które mogą być dziedziczone przez podklasy wliczając w to kolejność pokazywania na ekranie (malowania), widoczność, przekształcenia, obsługę stylów CSS i wiele wiele innych. Klasy bezpośrednio dziedziczące z klasy Parent to np. Control, Group, Region, WebView. Klasy węzłów liści są zdefiniowane w innych, dodatkowych pakietach np. `javafx.scene.shape` lub `javafx.scene.text`.

Utwórz nowy projekt oparty o Maven'a, następnie w pliku `pom.xml` dodaj odpowiednie zależności (istotne zostały pogrubione).

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>org.example</groupId>
  <artifactId>GUIFX</artifactId>
  <version>1.0-SNAPSHOT</version>

  <properties>
    <maven.compiler.source>14</maven.compiler.source>
    <maven.compiler.target>14</maven.compiler.target>
  </properties>
  <dependencies>
    <dependency>
      <groupId>org.openjfx</groupId>
      <artifactId>javafx-graphics</artifactId>
      <version>18.0.2</version>
    </dependency>

    <dependency>
      <groupId>org.openjfx</groupId>
      <artifactId>javafx-controls</artifactId>
      <version>18.0.2</version>
    </dependency>

    <dependency>
      <groupId>org.openjfx</groupId>
      <artifactId>javafx-fxml</artifactId>
      <version>18.0.2</version>
```



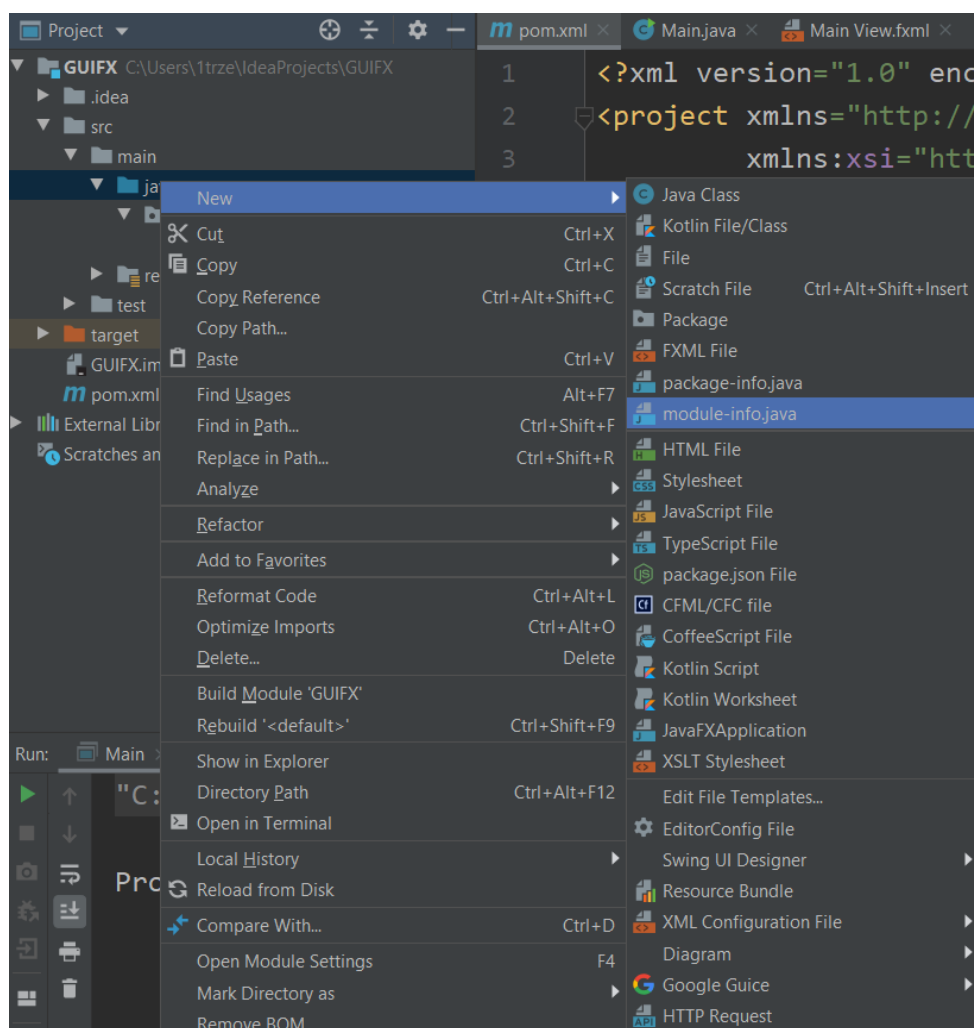
```
</dependency>

</dependencies>

</project>
```

Każda z tych zależności odwołuje się do innej części całego pakietu JavaFX, wszystkie możliwe są do śledzenia na stronie repozytorium Maven'a: <https://mvnrepository.com/search?q=JavaFX>.

W głównym katalogu źródłowym należy utworzyć plik module-info.java zgodnie z rysunkiem 7.5, a następnie wypełnić go odpowiednimi komendami.



Rys. 7.5. Tworzenie pliku module-info.java.

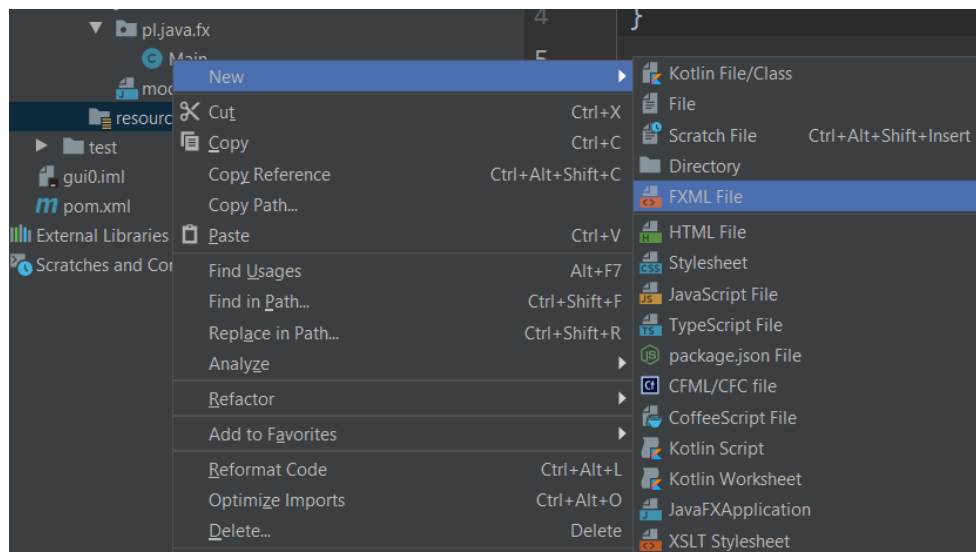
Warto zauważyć, że nazwa modułu jest taka sama jak nazwa projektu, który został utworzony. Wewnątrz pliku określone są wymagane moduły, które są niezbędne w projekcie (bo np. wykorzystywane są kontrolki) i bez tego można napotkać na wyjątek nieodnalezienia klasy, przykładowo:

```
Caused by: java.lang.ClassNotFoundException:  
javafx.scene.control.Button
```

jeżeli w projekcie jest wykorzystywany przycisk. Poniżej zawartość pliku module-info.java.

```
module GUIFX {  
  
    requires javafx.controls;  
    requires javafx.graphics;  
    requires javafx.fxml;  
  
    exports pl.java.fx to javafx.graphics;  
  
}
```

Kolejnym krokiem jest utworzenie pliku Main.java w pakiecie pl.java.fx. Ze względu na możliwość tworzenia elementów graficznego interfejsu użytkownika przy pomocy kodu XML w katalogu resources należy utworzyć plik FXML New → FXML File → MainView.fxml.



Rys. 7.6. Tworzenie pliku FXML.

Edytowanie takiego pliku FXML jest możliwe na dwa sposoby:

- ręczne tworzenie kodu XML w edytorze tekstowym,
- wykorzystanie specjalnego narzędzia Scene Builder (w środowisku IDE lub w oddzielnej aplikacji).

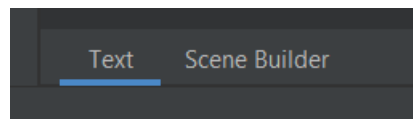
```
<?xml version="1.0" encoding="UTF-8"?>  
  
<?import java.lang.*?>  
<?import java.util.*?>  
<?import javafx.scene.*?>
```

```
<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>

<AnchorPane xmlns="http://javafx.com/javafx"
             xmlns:fx="http://javafx.com/fxml"
             fx:controller="MainView"
             prefHeight="400.0" prefWidth="600.0">

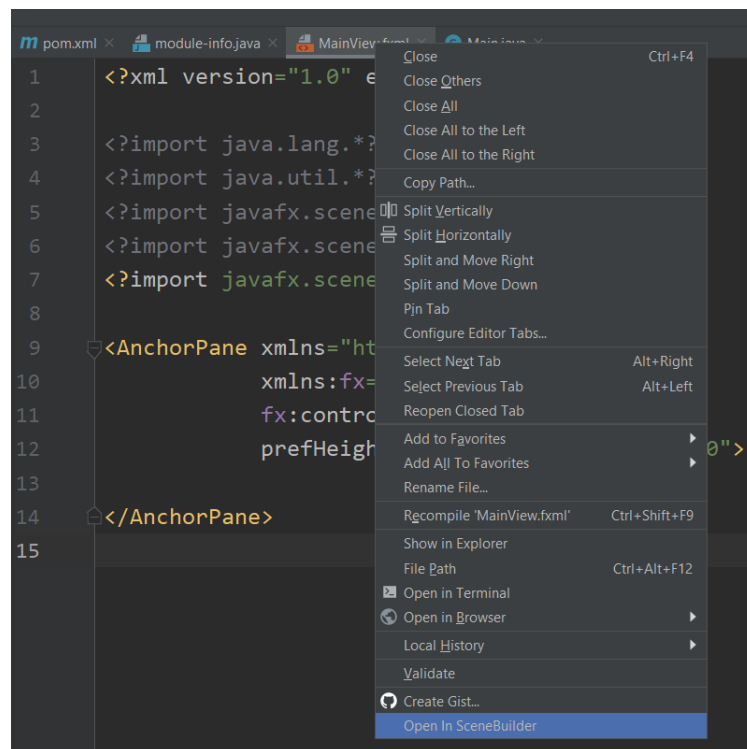
</AnchorPane>
```

Do przełączenia między edytorem tekstowym, a Scene Builder'em służy zakładka na dole strony, po otwarciu pliku *.fxml.



Rys. 7.7. Edytor tekstowy lub Scene Bulder.

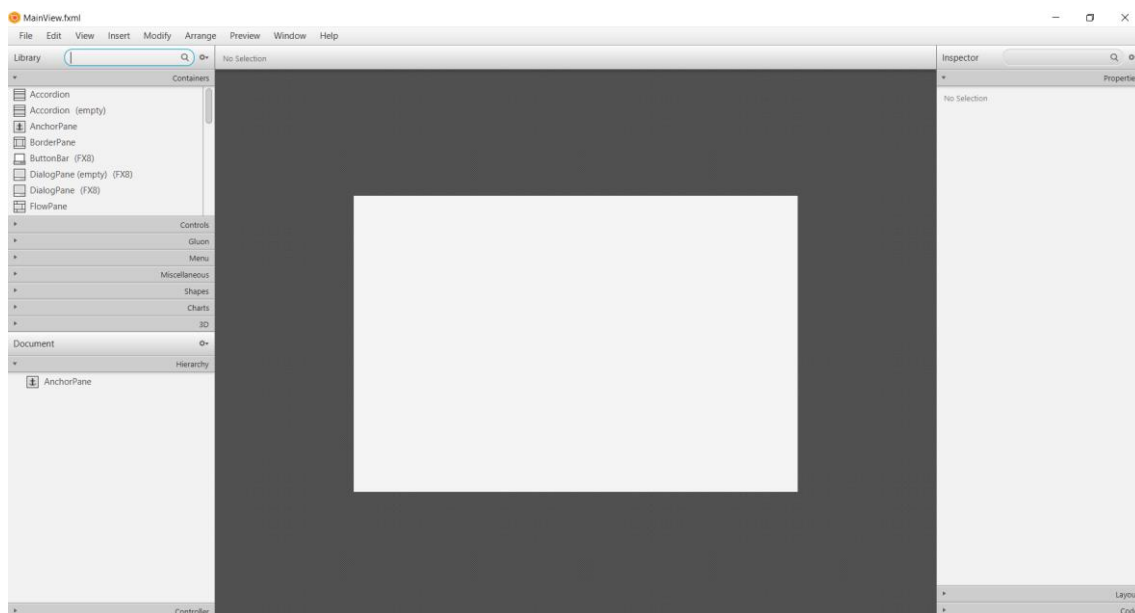
Dodatkowym atutem jest możliwość otwarcia narzędzia Scene Builder niezależnie od środowiska programistycznego. Na pliku *.fxml należy kliknąć PPM → Open In SceneBuilder, jak pokazano na rysunku 7.8.



Rys. 7.8. Otwieranie pliku w zewnętrznej aplikacji.

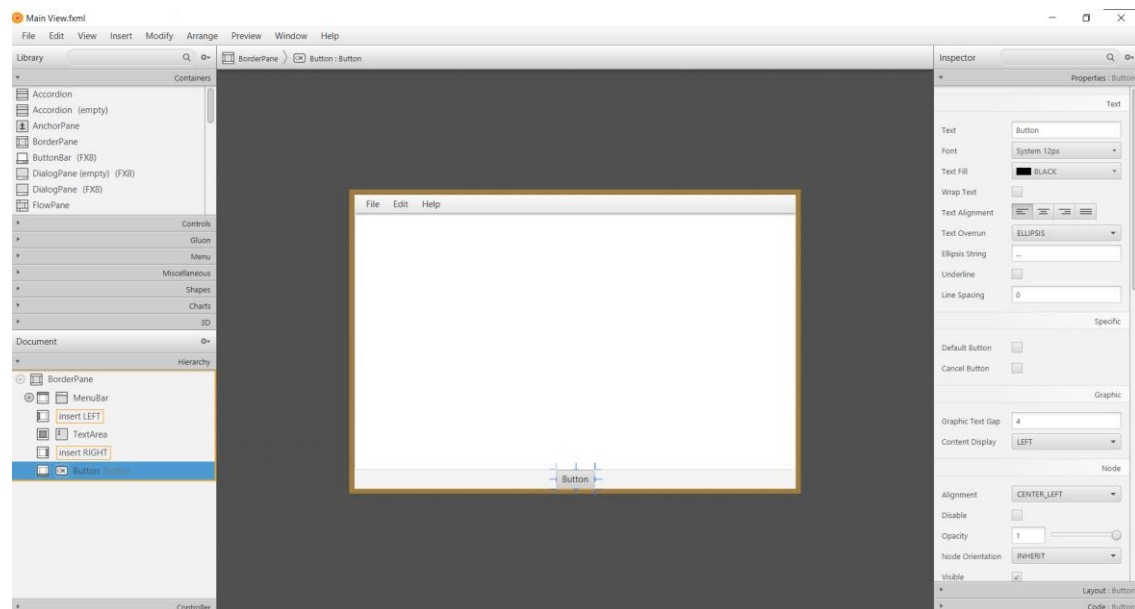
Scene Builder jest bardzo intuicyjnym narzędziem, które zdecydowanie pomaga tworzyć aplikacje GUI. Po lewej stronie znajduje się lista kontenerów, komponentów i innych

przypadkowych elementów. Metodą przeciągnij i upuść należy dodawać poszczególne elementy, a równolegle kod pliku uzupełnia się automatycznie.



Rys. 7.9. Podstawowy widok Scene Builder 'a'.

Przykładowo interfejs użytkownika może składać się z wielolinijkowego pola tekstowego, przycisku oraz menu górnego, jak pokazano na rysunku 7.10



Rys. 7.10. Dodawanie elementów do sceny.

Kod XML dla tak przygotowanego okna zostanie wygenerowany automatycznie (oczywiście możliwa jest modyfikacja).

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.control.Button?>
<?import javafx.scene.control.Menu?>
<?import javafx.scene.control.MenuBar?>
<?import javafx.scene.control.MenuItem?>
<?import javafx.scene.control.TextArea?>
<?import javafx.scene.layout.BorderPane?>

<BorderPane maxHeight="-Infinity" maxWidth="-Infinity"
minHeight="-Infinity" minWidth="-Infinity" prefHeight="400.0"
prefWidth="600.0" xmlns="http://javafx.com/javafx/18"
xmlns:fx="http://javafx.com/fxml/1">
    <center>
        <TextArea prefHeight="200.0" prefWidth="200.0"
BorderPane.alignment="CENTER" />
    </center>
    <bottom>
        <Button mnemonicParsing="false" text="Button"
BorderPane.alignment="CENTER" />
    </bottom>
    <top>
        <MenuBar BorderPane.alignment="CENTER">
            <menus>
                <Menu mnemonicParsing="false" text="File">
                    <items>
                        <MenuItem mnemonicParsing="false" text="Close"
/>
                    </items>
                </Menu>
                <Menu mnemonicParsing="false" text="Edit">
                    <items>
                        <MenuItem mnemonicParsing="false" text="Delete"
/>
                    </items>
                </Menu>
                <Menu mnemonicParsing="false" text="Help">
                    <items>
                        <MenuItem mnemonicParsing="false" text="About"
/>
                    </items>
                </Menu>
            </menus>
        </MenuBar>
    </top>
</BorderPane>
```


Zmiana elementów lub nawet ich rozmieszczenia może wpłynąć na ten kod. Odpowiada on tylko i wyłącznie za widok. W tym miejscu należy zastanowić się nad pozostałą częścią aplikacji. W jaki sposób połączyć widok z logiką? Dobrym rozwiązaniem jest zastosowanie wzorca projektowego (w tym przypadku będzie to MVC).

Wzorce projektowe to rozwiązanie wielu problemów przy pomocy uniwersalnego sposobu. Nie jest to gotowy kod ale raczej przepis na zaplanowanie i wykonanie krok po kroku określonego schematu dopasowując go jednocześnie do narzuconych wymagań. Wzorec MVC jest jednym z najczęściej stosowanych wzorców projektowych w informatyce. Jego idee można zrozumieć rozszyfrowując nazwę: Model-View-Controller (Model-Widok-Kontroler). Głównym założeniem tego wzorca jest podzielenie kodu aplikacji na 3 moduły:

- model – przedstawienie danych,
- widok – przedstawienie interfejsu użytkownika,
- kontroler – logika aplikacji.

Model zapewnia jednorondy sposób dostępu do danych. Najczęściej stosowany jest do pobierania i przygotowania rekordów z bazy danych (ale nie jest to jedyne źródło danych). Dzięki temu reszta aplikacji staje się niezależna od tego skąd i w jaki sposób pobierane są dane. Kontroler (czyli nasza logika) nie musi wiedzieć czy pobieramy dane z bazy (a jeśli tak jest to nie musi wiedzieć czy korzystamy np. z MySQL czy PostgreSQL) czy np. z pliku. Pisząc logikę aplikacji wywołujemy tylko odpowiednie funkcje modelu i w wyniku dostajemy dane do przetworzenia. W połowie tworzenia programu/strony typ i format składowania danych może się zmienić ale będzie to wymagać zmiany kodu tylko modelu. Model jest opcjonalny gdyż nie zawsze korzystamy z danych pobieranych z bazy czy plików.

Widok reprezentuje to co widzi użytkownik. Znowu powstaje pytanie: dlaczego nie zawrzeć tego w kodzie aplikacji? Widok oddzielony od logiki pozwala na bezinwazyjną zmianę grafiki w dowolnym momencie. Kontroler musi jedynie wiedzieć w jaki sposób przekazać dane do widoku. Osoba wykonująca widok nie musi wiedzieć praktycznie nic o logice programu gdyż dostaje tylko dane, które trzeba sformatować. Należy również podkreślić, że – tak jak w przypadku modelu – widok może się zmienić w każdej chwili nie ingerując w kod programu. Widok może być zminimalizowany (np. do linii poleceń).

Kontroler jest podstawową jednostką logiczną naszego programu. To tutaj znajduje się najważniejsza część kodu. Kontroler odpowiada m.in. za przetwarzanie danych pobranych za pomocą modelu i przekazanie ich użytkownikowi oraz zapisanie danych przez niego podanych (poprzez widok). W kontrolerze odbywają się wszystkie konieczne obliczenia i podejmowane są odpowiednie akcje w zależności od działań użytkownika. Krótko mówiąc kontroler zajmuje się sterowaniem całą aplikacją i jest jej najważniejszym elementem. Z powodu przydatności jest to prawdopodobnie najczęściej wykorzystywany wzorec projektowy.

Poniżej zaprezentowano przykładowy projekt oparty o wzorec MVC. Prosta aplikacja pokazująca w jaki sposób należy połączyć ze sobą moduły. Jak każda prosta aplikacja może zostać modyfikowana i ulepszana ale dzięki zastosowaniu wzorca i oddzieleniu widoku od logiki można zmieniać dowolnie logikę lub wygląd bez obaw o pojawiające się błędy (oczywiście należy zachować odpowiedni układ). Projekt został pobrany ze strony: <https://github.com/edencoding/javafx-core> i zawiera dwie wersje, w oparciu o FXML oraz z połączeniem tradycyjnym (tylko przy pomocy kodu Java).

Plik pom.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.edencoding</groupId>
  <artifactId>MVCEExample</artifactId>
  <version>1.0-SNAPSHOT</version>

  <properties>
    <project.build.sourceEncoding>UTF-
8</project.build.sourceEncoding>
    <maven.compiler.source>11</maven.compiler.source>
    <maven.compiler.target>11</maven.compiler.target>
    <javafx.version>16</javafx.version>
  </properties>
  <dependencies>
    <dependency>
      <groupId>org.openjfx</groupId>
      <artifactId>javafx-controls</artifactId>
      <version>${javafx.version}</version>
    </dependency>
    <dependency>
      <groupId>org.openjfx</groupId>
      <artifactId>javafx-swing</artifactId>
      <version>${javafx.version}</version>
    </dependency>
    <dependency>
      <groupId>org.openjfx</groupId>
      <artifactId>javafx-fxml</artifactId>
      <version>${javafx.version}</version>
    </dependency>
  </dependencies>
</project>
```

Plik module-info.java:

```
module com.edencoding {
  requires javafx.controls;
  requires javafx.fxml;
  requires java.desktop;

  opens com.edencoding.controllers to javafx.fxml;
```



Fundusze Europejskie
Wiedza Edukacja Rozwój



**Rzeczpospolita
Polska**

Unia Europejska
Europejski Fundusz Społeczny



```
exports com.edencoding;}
```

Plik FinancialAccount.java:

```
package com.edencoding.models;

import javafx.beans.property.*;

public class FinancialAccount {

    private final StringProperty accountHolder;
    private final IntegerProperty accountNumber;
    private final DoubleProperty accountBalance;

    public FinancialAccount(String accountHolder, Integer
accountNumber, Double accountBalance) {
        this.accountHolder = new
SimpleStringProperty(accountHolder);
        this.accountNumber = new
SimpleIntegerProperty(accountNumber);
        this.accountBalance = new
SimpleDoubleProperty(accountBalance);
    }

    public String getAccountHolder() {
        return accountHolder.get();
    }

    public StringProperty accountHolderProperty() {
        return accountHolder;
    }

    public int getAccountNumber() {
        return accountNumber.get();
    }

    public IntegerProperty accountNumberProperty() {
        return accountNumber;
    }

    public double getAccountBalance() {
        return accountBalance.get();
    }

    public DoubleProperty accountBalanceProperty() {
        return accountBalance;
    }

    public void deposit(double amount){
```



```
        accountBalance.set(accountBalance.get() + amount);
    }

    public void withdraw(double amount){
        accountBalance.set(accountBalance.get() - amount);
    }
}
```

Plik `mainView.fxml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<?import javafx.geometry.Insets?>
<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>
<VBox xmlns="http://javafx.com/javafx/10.0.2-internal"
xmlns:fx="http://javafx.com/fxml/1"
    prefHeight="250.0" prefWidth="300.0"
    spacing="15.0"
    stylesheets="@../css/styles.css"

fx:controller="com.edencoding.controllers.MainViewController">
    <Label alignment="CENTER" maxWidth="600"
styleClass="title" text="Bank Account" />
    <GridPane>
        <columnConstraints>
            <ColumnConstraints hgrow="SOMETIMES"/>
            <ColumnConstraints hgrow="SOMETIMES"/>
        </columnConstraints>
        <rowConstraints>
            <RowConstraints minHeight="10.0" prefHeight="30.0"
vgrow="SOMETIMES"/>
            <RowConstraints minHeight="10.0" prefHeight="30.0"
vgrow="SOMETIMES"/>
            <RowConstraints minHeight="10.0" prefHeight="30.0"
vgrow="SOMETIMES"/>
        </rowConstraints>
        <Label styleClass="bold" text="Account holder:"/>
        <Label fx:id="accountHolder"
GridPane.columnIndex="1"/>
        <Label styleClass="bold" text="Account Number:"
GridPane.rowIndex="1"/>
        <Label fx:id="accountNumber" GridPane.columnIndex="1"
GridPane.rowIndex="1"/>
        <Label styleClass="bold" text="Balance:"
GridPane.rowIndex="2"/>
        <Label fx:id="accountBalance" GridPane.columnIndex="1"
GridPane.rowIndex="2"/>
    </GridPane>
    <HBox alignment="CENTER" spacing="25.0">
```



```
        <Button onAction="#handleWithdrawal" text="Withdraw"/>
        <TextField fx:id="amountTextField" prefWidth="75.0"
promptText="Number"/>
        <Button layoutX="10.0" layoutY="10.0"
onAction="#handleDeposit" text="Deposit"/>
    </HBox>
    <padding>
        <Insets topRightBottomLeft="15" />
    </padding>
</VBox>
```

Plik MainViewController.java:

```
package com.edencoding.controllers;

import com.edencoding.models.FinancialAccount;
import javafx.event.Event;
import javafx.fxml.FXML;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.control.TextFormatter;

public class MainViewController {

    //Model
    FinancialAccount account;

    //View nodes
    @FXML private Label accountHolder;
    @FXML private Label accountNumber;
    @FXML private Label accountBalance;
    @FXML private TextField amountTextField;

    public void initialize(){
        //get model
        account = new FinancialAccount("Maxwell Planck", 6626,
1000d);

        //link Model with View

accountHolder.textProperty().bind(account.accountHolderPropert
y());

accountBalance.textProperty().bind(account.accountBalancePrope
rty().asString());

accountNumber.textProperty().bind(account.accountNumberPropert
y().asString());
```



```
//link Controller to View - ensure only numeric input
(integers) in text field
amountTextField.setTextFormatter(new
TextFormatter<>(change -> {
    if (change.getText().matches("\\d+") ||
change.getText().equals("")) {
        return change;
    } else {
        change.setText("");
        change.setRange(
            change.getRangeStart(),
            change.getRangeStart()
        );
        return change;
    }
}));

@FXML private void handleDeposit(Event event) {
    account.deposit(getAmount());
    event.consume();
}

@FXML private void handleWithdrawal(Event event) {
    account.withdraw(getAmount());
    event.consume();
}

private double getAmount(){
    if (amountTextField.getText().equals("")) return 0;

    return Double.parseDouble(amountTextField.getText());
}
}
```

Plik MVCExampleApp.java:

```
package com.edencoding;

import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.scene.image.Image;
import javafx.stage.Stage;

public class MVCExampleApp extends Application {

    @Override
```



Fundusze Europejskie
Wiedza Edukacja Rozwój



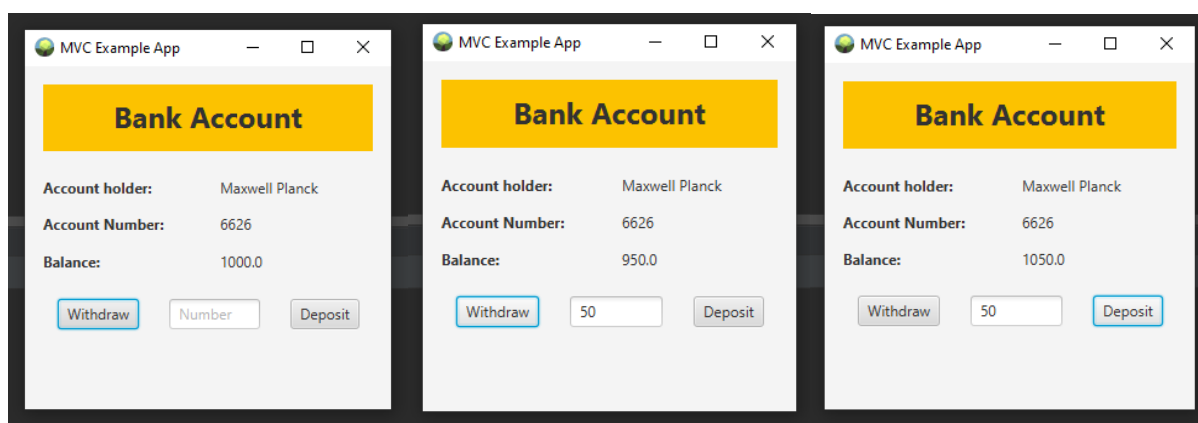
**Rzeczpospolita
Polska**

Unia Europejska
Europejski Fundusz Społeczny



```
public void start(Stage primaryStage) throws Exception{
    Parent root =
FXMLLoader.load(getClass().getResource("/fxml/mainView.fxml"))
;
    primaryStage.setTitle("MVC Example App");
    primaryStage.getIcons().add(new
Image(getClass().getResource("/img/EdenCodingIcon.png").toExternalForm()));
    primaryStage.setScene(new Scene(root, 300, 275));
    primaryStage.show();
}

public static void main(String[] args) {
    launch(args);
}
}
```



Rys. 7.11. Przykładowy widok aplikacji.

Zastosowanie metody `setTextFormatted()` wewnątrz metody inicjalizującej powoduje, że niemożliwe jest umieszczenie w polu tekstowym czegokolwiek innego niż cyfry (z których składa się liczba). Jak zostało to wcześniej zauważone jest to prosta aplikacja, więc na pierwszy rzut oka można wprowadzić usprawnienia. Wykonaj poniższe zadania.

Zadanie 7.1. Drobne usprawnienia.

Dodaj do projektu:

- możliwość wprowadzania wartości rzeczywistej – zwróć uwagę na poprawny separator,
- oprocentowanie depozytu (np. w formie etykiety lub suwaka bez możliwości edycji) – wartość oprocentowania (pozycja suwaka) ma zmieniać się wraz ze zmianą wartości depozytu,
- alarm debetowy – po przekroczeniu ustalonej wartości (np. -5000) następuje blokada możliwości wypłat z dodatkową informacją (w formie wyskakującego okienka),

Zadanie 7.2. Modyfikacja projektu.

Rozważ możliwość dodawania kolejnych kont. Czy jest to możliwe bez większych ingerencji w projekt? Przygotuj klasy pośredniczące jeśli jest to niezbędne.

Zadanie 7.3. Źródło danych.

Rozbuduj aplikację o możliwość zapisywania i odczytywania danych dla różnych kont. Źródłem może być plik tekstowy lub baza danych. Zadbaj o wymagania z zadania 7.1.

Zadanie 7.4. Aplikacja GUI

Napisz prostą aplikację zawierającą graficzny interfejs użytkownika. Przykłady aplikacji:

- kalkulator,
- kalkulator walutowy,
- prosty edytor tekstowy,
- kreślarka wykresów,
- notatnik z pamięcią.

Zadanie 7.5. Aplikacja GUI – baza danych

Zaprojektuj i wykonaj aplikację z graficznym interfejsem użytkownika. Celem aplikacji jest wykonywanie operacji CRUD (Create Read Update Delete) na bazie danych przechowującej dane studentów. Zastosuj wzorzec MVC. Model studenta powinien zawierać dane obowiązkowe i dodatkowe. Przykładowe dane obowiązkowe to:

- imię,
- nazwisko,
- nr indeksu,
- adres,
- kierunek studiów/specjalność,
- rok studiów/semestr.

Dane dodatkowe:

- dowolne inne dane (np. średnia za ostatni semestr, czy przyznano stypendium itp...).

Aplikacja GUI powinna mieć możliwość dodania danych studenta do bazy, odczytania danych istniejących w bazie studentów, modyfikacji danych i usunięcia danych. W przypadku pustej bazy (brak danych) należy wyświetlić odpowiedni komunikat. Przy dodawaniu i modyfikacji niezbędne jest podanie wszystkich poprawnie sformatowanych danych obowiązkowych(!). Dane dodatkowe powinny być walidowane ale nie są niezbędne.

LABORATORIUM 8. TWORZENIE APLIKACJI Z WYKORZYSTANIEM BIBLIOTEKI SPRING.

Cel laboratorium:

Zapoznanie studentów z biblioteką Spring. Tworzenie aplikacji z wykorzystaniem biblioteki Spring.

Zakres tematyczny zajęć:

- Platforma Spring.
- Wybrane projekty Spring.
- Wybrane adnotacje.
- Tworzenie aplikacji w oparciu o Spring Boot'a.

Pytania kontrolne:

1. Czym jest platforma Spring? Jaka jest różnica między projektem i modulem?
2. Do jakich celów została utworzona platforma Spring?
3. W jaki sposób można wykorzystać projekty Spring?
4. Jakie znasz adnotacje występujące w modułach Spring?
5. Do czego służy serwis <https://start.spring.io>? Jakie są plusy jego wykorzystania?

Platforma Spring

Spring to obecnie najbardziej popularna platforma Java na świecie, której używa zdecydowana większość firm tworzących oprogramowanie. Umożliwia budowanie aplikacji Java w oparciu o wbudowane mechanizmy, które wspierają programistę w kluczowych kwestiach, takich jak tworzenie obiektów, przechowywanie ich i udostępnianie we właściwym momencie. Ułatwia obsługę żądań HTTP, w tym pozwala na wykorzystanie potencjału koncepcji REST. Dostarcza mechanizmy do obsługi baz danych, które opierają się na wykorzystaniu dostawcy JPA - najczęściej Hibernate'a. Platforma Spring zapewnia wszechstronny model programowania i konfiguracji dla nowoczesnych aplikacji korporacyjnych opartych na języku Java - na dowolnej platformie wdrożeniowej. Ze względu na ogromny rozrost i liczne elementy bibliotekę Spring możemy podzielić na projekty:

- Spring Boot,
- Spring Framework,
- Spring Data,
- Spring Security,
- Spring Cloud,
- i wiele wiele innych.

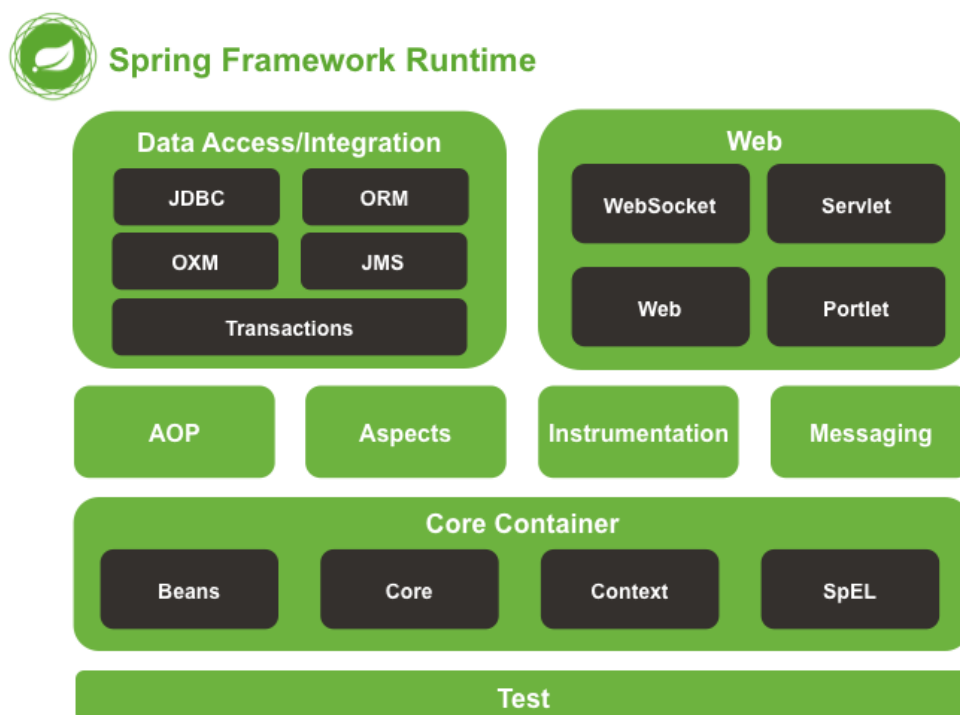
W pierwszej kolejności zostaną po krótko omówione wybrane projekty z całości platformy Spring.

Spring Boot

Ten projekt Spring'a ułatwia szybkie tworzenie samodzielnych aplikacji opartych na szkieletcie Spring, które można „po prostu uruchomić”. Większość aplikacji Spring Boot wymaga minimalnej konfiguracji Spring. Są to więc aplikacje gotowe od ręki.

Spring Framework

Podstawowy element platformy Spring, w oparciu o który można bardzo łatwo budować aplikacje webowe. Zapewnia wszechstronny model programowania i konfiguracji dla nowoczesnych aplikacji korporacyjnych opartych na języku Java - na dowolnej platformie wdrożeniowej. Kluczowym elementem Spring jest wsparcie infrastrukturalne na poziomie aplikacji: Spring koncentruje się na „instalacji” aplikacji korporacyjnych, tak aby zespoły mogły skupić się na logice biznesowej na poziomie aplikacji, bez zbędnych powiązań z określonymi środowiskami wdrożeniowymi. Spring Framework składa się z funkcji zorganizowanych w około 20 modułów. Moduły te są pogrupowane w podstawowe kontenery, dostęp do danych/integrację, sieć, AOP (programowanie zorientowane na aspekty), oprzyrządowanie, przesyłanie wiadomości i testowanie, jak pokazano na poniższym diagramie (źródło: <https://docs.spring.io/spring-framework/docs/4.3.x/spring-framework-reference/html/overview.html>).



Rys. 8.1. Podział Spring Framework.

Poniżej przedstawiono wybrane elementy projektu Spring Framework.

Spring Core

Kontener podstawowy składa się z modułów `spring-core`, `spring-beans`, `spring-context`, `spring-context-support` i `spring-expression` (Spring Expression Language).

Moduły `spring-core` i `spring-beans` zapewniają podstawowe części struktury, w tym funkcje IoC i Dependency Injection. `BeanFactory` to wyrafinowana implementacja wzorca fabryki. Eliminuje potrzebę programowych singletonów i pozwala oddzielić konfigurację i specyfikację zależności od rzeczywistej logiki programu.

Moduł `Context` (`spring-context`) opiera się na solidnej podstawie dostarczonej przez moduły `Core` i `Beans`: jest to sposób na dostęp do obiektów w sposób podobny do rejestru JNDI. Moduł `Context` dziedziczy swoje funkcje z modułu `Beans` i dodaje obsługę internacjonalizacji (przy użyciu np. pakietów zasobów), propagacji zdarzeń, ładowania zasobów oraz przejrzystego tworzenia kontekstów przez np. kontener `Servlet`. Moduł `Context` obsługuje również funkcje Java EE, takie jak EJB, JMX i podstawowe usługi zdalne. Interfejs `ApplicationContext` jest centralnym punktem modułu `Context`. Moduł `spring-context-support` zapewnia obsługę integracji popularnych bibliotek innych firm z kontekstem aplikacji Spring w celu buforowania (`EhCache`, `Guava`, `JCache`), wysyłania poczty (`JavaMail`), planowania (`CommonJ`, `Quartz`) i silników szablonów (`FreeMarker`, `JasperReports`, `Velocity`).

Moduł `spring-expression` zapewnia potężny język wyrażeń do wykonywania zapytań i manipulowania wykresem obiektów w czasie wykonywania. Jest to rozszerzenie zunifikowanego języka wyrażeń (`unified EL`) określonego w specyfikacji JSP 2.1. Język obsługuje ustawianie i pobieranie wartości właściwości, przypisywanie właściwości, wywoływanie metod, dostęp do zawartości tablic, kolekcji i indeksatorów, operatorów logicznych i arytmetycznych, nazwanych zmiennych oraz pobieranie obiektów według nazwy z kontenera IoC Springa. Obsługuje również projekcję i wybór list, a także wspólne agregacje list.

Aspect-oriented programming

Moduł `spring-aop` zapewnia implementację programowania zorientowanego aspektowo zgodną z AOP Alliance, umożliwiającą zdefiniowanie, na przykład przechwytywaczy metod i punktów przecięcia w celu czystego oddzielenia kodu, który implementuje funkcjonalność, która powinna być rozdzielona. Korzystając z funkcji metadanych na poziomie źródła, można również włączyć informacje behawioralne do kodu w sposób podobny do atrybutów platformy .NET. Oddzielny moduł `AspectJ` zapewnia integrację z `AspectJ`.

Moduł `spring-instrument` zapewnia obsługę instrumentacji klas i implementacje klas ładujących, które mogą być używane w niektórych serwerach aplikacji. Moduł `spring-instrument-tomcat` zawiera agenta oprzyrządowania Springa dla Tomcata.

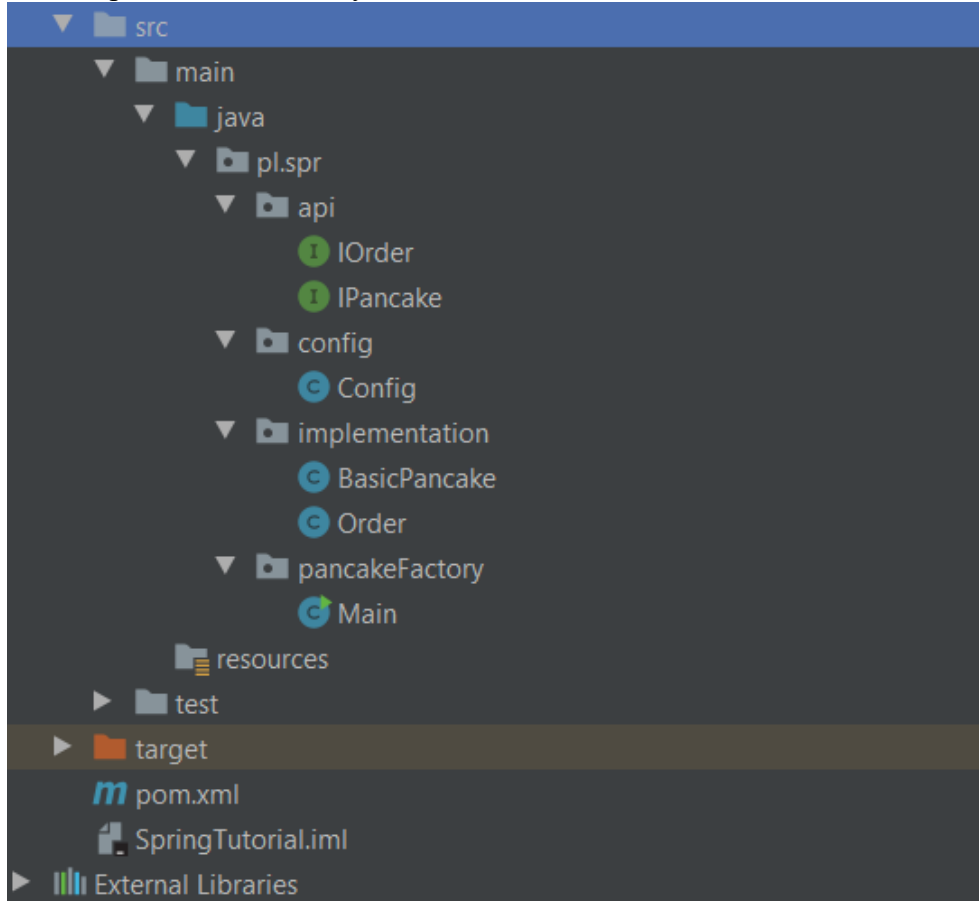
Testy

Moduł `spring-test` obsługuje testy jednostkowe i testy integracyjne komponentów Spring z JUnit lub TestNG. Zapewnia spójne ładowanie Spring



ApplicationContexts i buforowanie tych kontekstów. Zapewnia również pozorowane obiekty, których można użyć do testowania kodu w izolacji.

Poniższy kod został przygotowany jako przykład zastosowania modułu spring – context do wstrzykiwania zależności w prostej aplikacji do zamawiania naleśników. Struktura aplikacji została przedstawiona na rysunku 8.2.



Rys. 8.2. Struktura aplikacji opartej na Spring.

Kod kolejnych plików został przedstawiony poniżej.

Kod interfejsu IOrder:

```
package pl.spr.api;

public interface IOrder {
    void printOrder();
}
```

Kod interfejsu IPancake:

```
package pl.spr.api;

public interface IPancake {
```

```
double getPrice();  
String getName();  
}
```

Kod klasy Order:

```
package pl.spr.implementation;  
  
import pl.spr.api.IOrder;  
import pl.spr.api.IPancake;  
  
public class Order implements IOrder {  
  
    private IPancake pancake;  
  
    public Order(IPancake pancake) {  
        this.pancake = pancake;  
    }  
  
    public void printOrder() {  
        System.out.println("Your order: " + pancake.getName()+  
            "\n"+"Price: " + pancake.getPrice());  
    }  
}
```

Kod klasy BasicPancake:

```
package pl.spr.implementation;  
  
import pl.spr.api.IPancake;  
  
public class BasicPancake implements IPancake {  
    private int price;  
    private String name;  
  
    public BasicPancake(int price, String name) {  
        this.price = price;  
        this.name = name;  
    }  
  
    public double getPrice() {  
        return price;  
    }  
  
    public String getName() {  
        return name;  
    }  
}
```



Kod pliku konfiguracyjnego Config:

```
package pl.spr.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import pl.spr.api.IOrder;
import pl.spr.api.IPancake;
import pl.spr.implementation.BasicPancake;
import pl.spr.implementation.Order;

@Configuration
public class Config {
    @Bean
    public IPancake pancake() {
        return new BasicPancake(20, "Podstawowy naleśnik");
    }

    @Bean
    public IOrder order(IPancake pancake) {
        return new Order(pancake);
    }
}
```

Kod pliku Main:

```
package pl.spr.pancakeFactory;

import org.springframework.context.annotation.AnnotationConfigApplica
tionContext;
import pl.spr.api.IOrder;
import pl.spr.config.Config;
import pl.spr.implementation.Order;

public class Main {
    public static void main(String[] args) {

        AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext(Config.class);
        IOrder order = context.getBean(Order.class);
        order.printOrder();
    }
}
```



Specjalne adnotacje pozwalają na utworzenie bean'a czyli specjalnego obiektu, który jest zarządzany przez kontekst Springa. Należą do nich m.in. adnotacja @Bean lub @Component.

W celu lepszego zapoznania się z dokumentacją spróbuj znaleźć różnicę między tymi dwiema adnotacjami (@Bean vs @Component).

Nie ma potrzeby tworzenia tych obiektów w sposób tradycyjny, ponieważ ich zarządcą jest Spring i on odpowiada za ich utworzenie. Rola programisty jest sprowadzona do określenia miejsca gdzie taki obiekt ma zostać utworzony. Jest to tzw. wstrzykiwanie zależności (ang. Dependency Injection, DI). Ten mechanizm polega na tym, że to framework odpowiada za cykl życia obiektów w aplikacji.

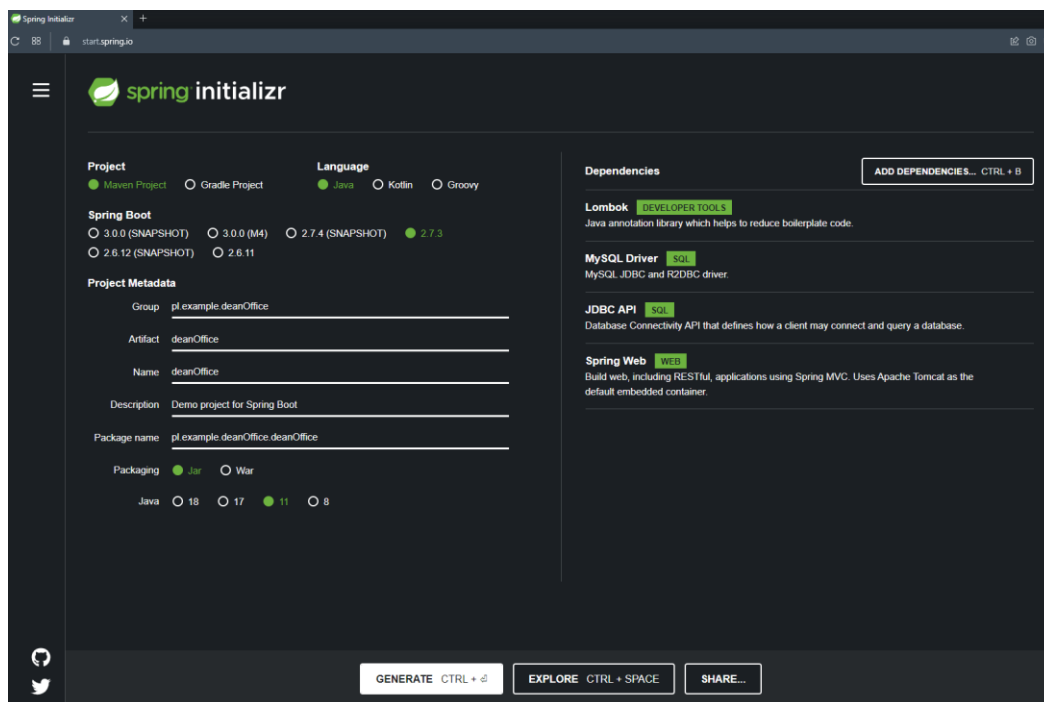
W metodzie main niezbędne jest utworzenie kontekstu dzięki czemu możliwe jest osadzenie utworzonych beanów w odpowiednich 'realiach'.

Tabela 8.1. Popularne adnotacje.

Adnotacja	Opis
@Component	ogólny, może być wykorzystywany w ramach definiowania beanów DTO
@Repository	dedykowana dla klas, których zadaniem jest przechowywanie, agregowanie danych
@Service	sugerowany dla klas, które dostarczają usługi
@Controller/@RestController	przeznaczony dla warstwy prezentacji lub/i dla API aplikacji

Kolejny przykład to aplikacja do wykonywania typowych operacji na bazie danych (CRUD). Do tego celu została stworzona aplikacja do zarządzania. Zostały tu wykorzystane następujące technologie: Spring Boot, MySQL, Docker. Wykorzystanie silnika bazodanowego MySQL oraz uruchomienie Dockera pozwala na połączenie kilku znanych już technologii w jednym projekcie.

Projekt rozpoczynamy od wygenerowania szablonu. Możliwe jest to na kilka sposobów, a ponieważ tradycyjnie projekt był tworzony w środowisku programistycznym IDE tym razem zostanie wykorzystany serwis <https://start.spring.io>. Jak pokazano na rysunku 8.3 należy wypełnić odpowiednie pola. Jest to o tyle wygodniejsze, że wszystkie niezbędne zależności wybieramy z listy i nie ma potrzeby oddzielnego wyszukiwania ich w repozytorium Mavena.



Rys. 8.3. Spring initializer.

W projekcie zostały dodane zależności od bibliotek: Lombok, MySQL Driver, JDBC API, oraz Spring Web. Po kliknięciu przycisku Generate na dole strony utworzone archiwum należy rozpakować, a następnie otworzyć w dowolnym środowisku IDE.

Pierwszą rzeczą jaką należy zrobić to próba uruchomienia tak załadowanego projektu. Można wykorzystać do tego zakładkę Maven (rys. 8.4) lub w terminalu wykonać polecenie:

```
mvn clean install
```



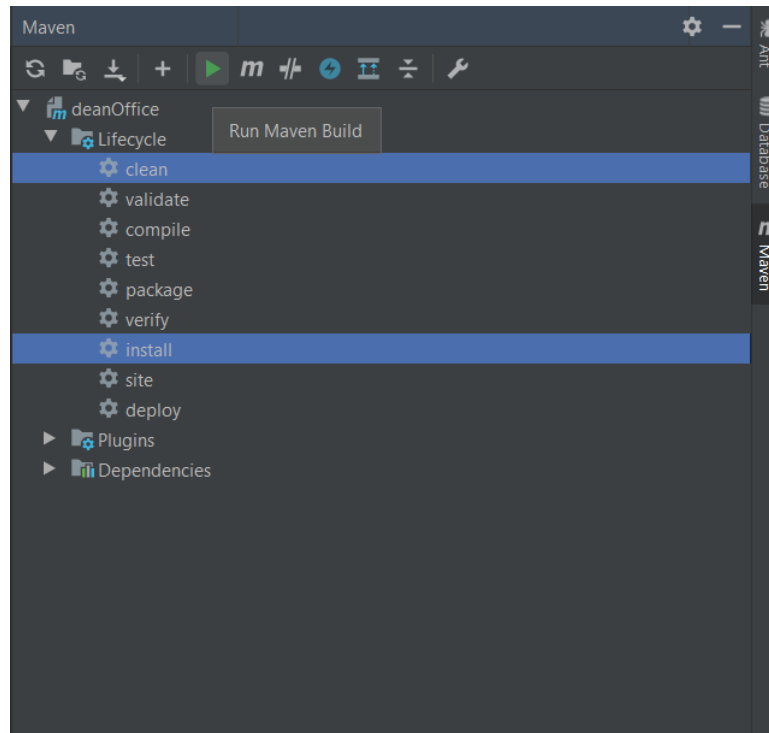
Fundusze Europejskie
Wiedza Edukacja Rozwój



**Rzeczpospolita
Polska**

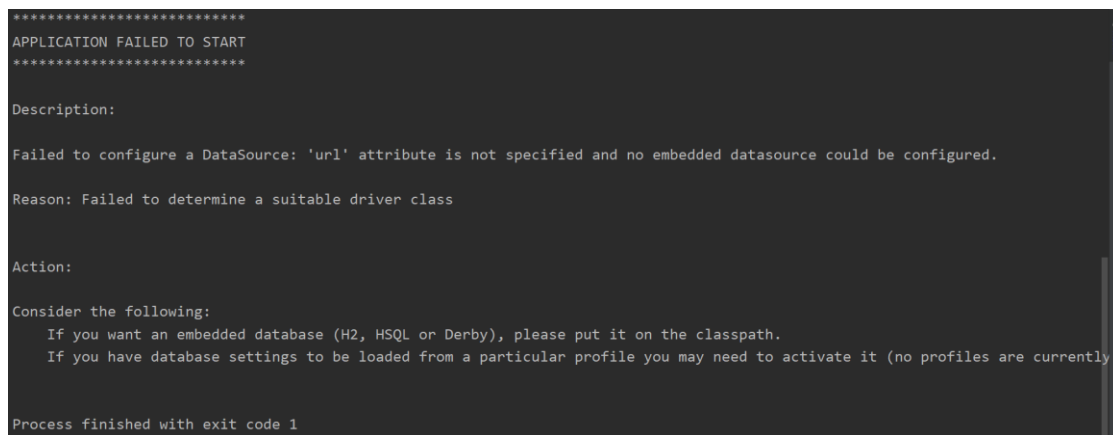
Unia Europejska
Europejski Fundusz Społeczny





Rys. 8.4. Uruchomienie procesu Maven'a – clean oraz install.

Bardzo szybko okazuje się, że w gotowym projekcie są braki, które powodują wystąpienie błędów – rysunek 8.5.



Rys. 8.5. Nieudana próba uruchomienia – brak atrybutu 'url' dla źródła danych.

Należy w katalogu resources odnaleźć plik application.properties i określić odpowiednie ustawienia konfiguracyjne dla źródła danych. Po dodaniu tych ustawień aplikacja może zostać uruchomiona.

```
spring.datasource.url =  
jdbc:mysql://localhost:3306/demobase?useSSL=false  
spring.datasource.username=root  
spring.datasource.password=root_password
```

Kolejnym krokiem jest utworzenie klasy modelowej odpowiadającej za ‘kształt’ danych oraz klasy odpowiadającej za kontroler. W klasie Student zawierającej podstawowe pola takie jak id, imię, nazwisko oraz średnia można dodać odpowiednie adnotacje z biblioteki Lombok w celu zmniejszenia linii kodu. Nazwy klas, pól i metod są w języku angielskim.

Kod klasy Student:

```
package pl.example.deanOffice.deanOffice;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@AllArgsConstructor
@NoArgsConstructor
public class Student {

    private int id;
    private String name;
    private String surname;
    private double averageGrade;

}
```

Następnie należy zdefiniować klasę StudentRepository, która będzie odpowiedzialna za komunikację z bazą danych. Odpowiednie adnotacje muszą być zastosowane, aby bezproblemowo Spring mógł zidentyfikować dany obszar np. jako klasę odpowiedzialną za agregowanie danych lub wykorzystanie metody do operacji bazodanowych.

Początkowy kod klasy StudentRepository:

```
package pl.example.deanOffice.deanOffice;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.BeanPropertyRowMapper;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Repository;

import java.util.List;

@Repository
public class StudentRepository {

    @Autowired
    JdbcTemplate jdbcTemplate;

}
```

```
public List<Student> getAll() {  
    return jdbcTemplate.query("SELECT id, name, surname,  
averageGrade",  
BeanPropertyRowMapper.newInstance(Student.class));  
}  
}
```

W tym miejscu należy skonfigurować i uruchomić bazę danych. Pierwszą rzeczą jest sprawdzenie, czy Docker jest zainstalowany (w wierszu poleceń) poleceniem:

```
docker -v
```

Jeżeli wszystko jest poprawnie zainstalowane powinna pokazać się dokładna wersja:

```
C:\Users\ltrze>docker -v  
Docker version 20.10.17, build 100c701
```

Rys. 8.6. Sprawdzenie wersji Docker'a.

Kolejnym poleceniem jest pobranie obrazu mysql (źródło: https://hub.docker.com/_/mysql):

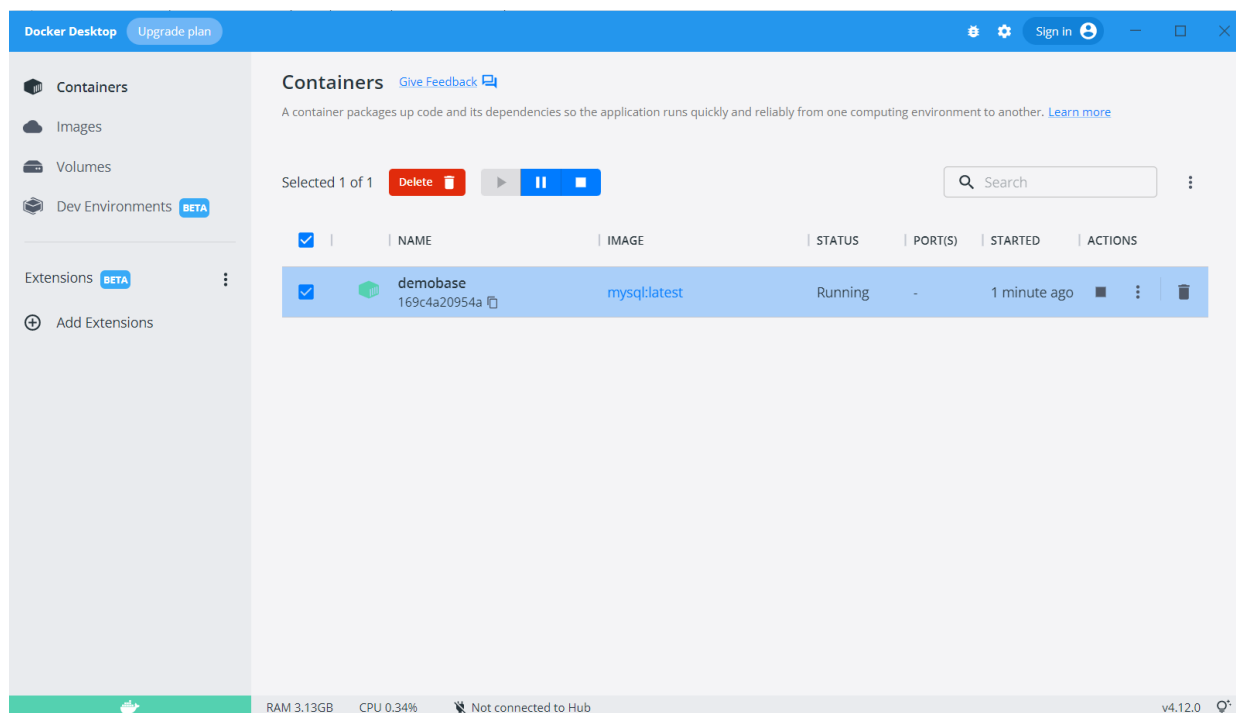
```
docker pull mysql
```

Po pobraniu wystarczy uruchomić instancję MySQL'a poleceniem:

```
docker run --name some-mysql -e MYSQL_ROOT_PASSWORD=my-secret-  
pw -d mysql:tag
```

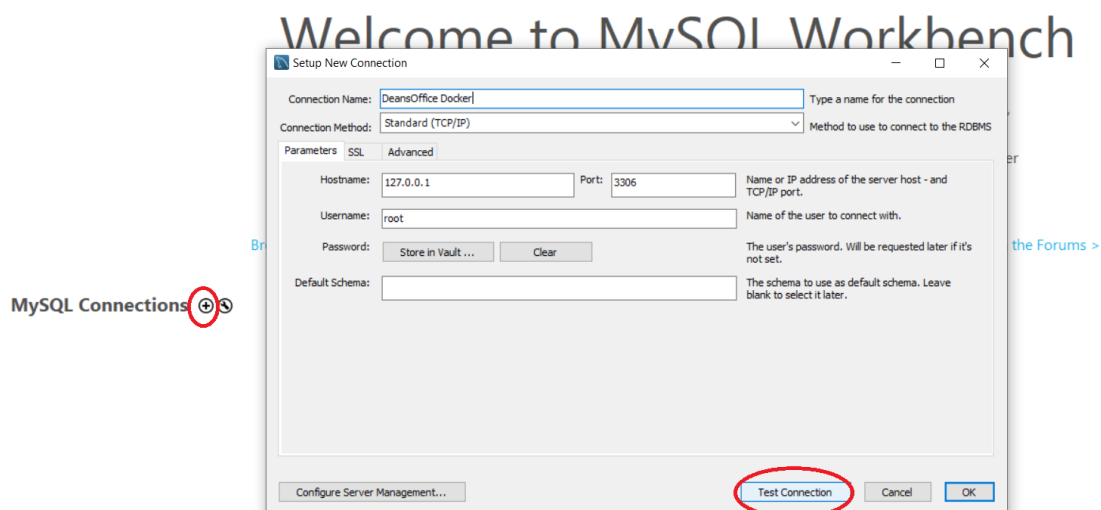
gdzie za wartości `some-mysql` oraz `my-secret-pw` należy podstawić odpowiednio nazwę bazy oraz hasło dla root'a określone wcześniej (`demobase` oraz `root_password`). Po poprawnym uruchomieniu bazy danych można ją podejrzeć w Dockerze lub w wierszu poleceń komendą:

```
docker ps
```

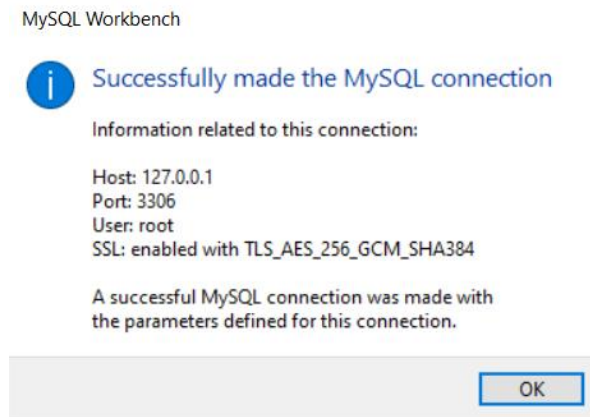


Rys. 8.7. Podgląd uruchomionego obrazu bazy danych.

Do zarządzania bazą danych zdecydowano się na MySQL Workbench. Konfiguracja połączenia jest bardzo intuicyjna. Należy określić nazwę połączenia, pozostałe ustawienia pozostawiamy domyślne i wystarczy po kliknięciu Test Connection podać hasło dla konta root. Po poprawnym utworzeniu połączenia powinno pokazać się powiadomienie widoczne na rysunku 8.9. Dalej należy już tylko otworzyć połączenie.

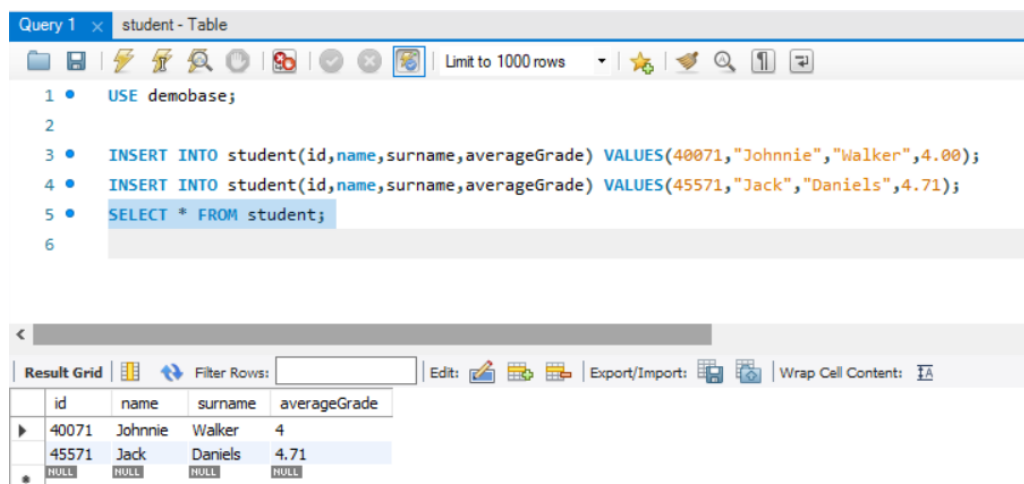


Rys. 8.8. Konfiguracja połączenia.



Rys. 8.9. Udany test połączenia.

Dodanie kilku studentów do bazy może okazać się pomocne przy testowaniu połączenia bazy z aplikacją.



Rys. 8.10. Dodanie przykładowych rekordów do bazy.

Ostatnim krokiem jest utworzenie klasy odpowiedzialnej za obsługę REST API. Odpowiednie adnotacje bardzo ułatwiają tworzenie kodu i redukują jego ilość. Klasa Controller będzie zawierała listę endpointów wykorzystujących metody wysyłające zapytania do bazy danych, a także odbierających odpowiedzi z bazy. Kod klasy Controller został przedstawiony poniżej:

```
package pl.example.deanOffice.deanOffice;  
  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.web.bind.annotation.*;  
  
import java.util.List;  
  
@RequestMapping("/student")  
@RestController
```

```
public class Controller {

    @Autowired
    StudentRepository studentRepository;

    @GetMapping("/")
    public List<Student> getAll(){
        return studentRepository.getAll();
    }

    @GetMapping("/{id}")
    public Student getById(@PathVariable("id") int id){
        return studentRepository.getById(id);
    }

    @PostMapping("")
    public int add(@RequestBody List<Student> students) {
        return studentRepository.save(students);
    }

    @PutMapping("/{id}")
    public int update(@PathVariable("id") int id, @RequestBody
Student updatedStudent) {
        Student student = studentRepository.getById(id);
        if (student != null) {
            student.setName(updatedStudent.getName());
            student.setSurname(updatedStudent.getSurname());

student.setAverageGrade(updatedStudent.getAverageGrade());
            studentRepository.update(student);
            return 1;
        } else {
            return -1;
        }
    }

    @PatchMapping("/{id}")
    public int partiallyUpdate(@PathVariable("id") int id,
@RequestBody Student updatedStudent) {
        Student student = studentRepository.getById(id);
        if (student != null) {

            if (updatedStudent.getName() != null)
student.setName(updatedStudent.getName());
            if (updatedStudent.getSurname() != null)
student.setSurname(updatedStudent.getSurname());
            if (updatedStudent.getAverageGrade() > 0.0)
student.setAverageGrade(updatedStudent.getAverageGrade());
            studentRepository.update(student);
        }
    }
}
```



```

        return 1;

    } else {
        return -1;
    }
}

@DeleteMapping("/{id}")
public int delete(@PathVariable("id") int id) {
    return studentRepository.delete(id);
}
}

```

Wyjaśnienie adnotacji znajduje się w tabeli 8.2.

Tabela 8.2. Adnotacje stosowane w klasie Controller.

Adnotacja	Opis
@RequestMapping	odpowiada za mapowanie punktów końcowych – użyte przed klasą doda zawartość przekazaną w nawiasie (np. @RequestMapping(„/java”) przed każdym endpointem wewnątrz klasy
@RestController	klasa przeznaczona do komunikacji dla API aplikacji – wartości zwracane przez metody będą konwertowane do postaci JSON lub XML
@Autowired	oznaczenie pozwala na wstrzykiwanie zależności w konstruktorze
@GetMapping	oznaczenie metody, która ma przyjąć żądanie typu GET
@PostMapping	oznaczenie metody, która ma przyjąć żądanie typu POST
@PutMapping	oznaczenie metody, która ma przyjąć żądanie typu PUT
@PatchMapping	oznaczenie metody, która ma przyjąć żądanie typu PATCH
@DeleteMapping	oznaczenie metody, która ma przyjąć żądanie typu DELETE
@PathVariable	obsługuje zmienne podczas mapowania i ustawiania ich jako parametr metody
@RequestBody	mapuje treść z żądania i umożliwia jej automatyczną deserializację do postaci obiektu Java

Uzupełnienie klasy StudentRepository po utworzeniu klasy Controller:

```

/*
...
*/

public Student getById(int id) {
    return jdbcTemplate.queryForObject("SELECT id, name,
surname, averageGrade FROM " +
        "student WHERE" + " id = ?",
        BeanPropertyRowMapper.newInstance(Student.class),
        id);
}

public int save(List<Student> students) {

```



Fundusze Europejskie
Wiedza Edukacja Rozwój



**Rzeczpospolita
Polska**

Unia Europejska
Europejski Fundusz Społeczny

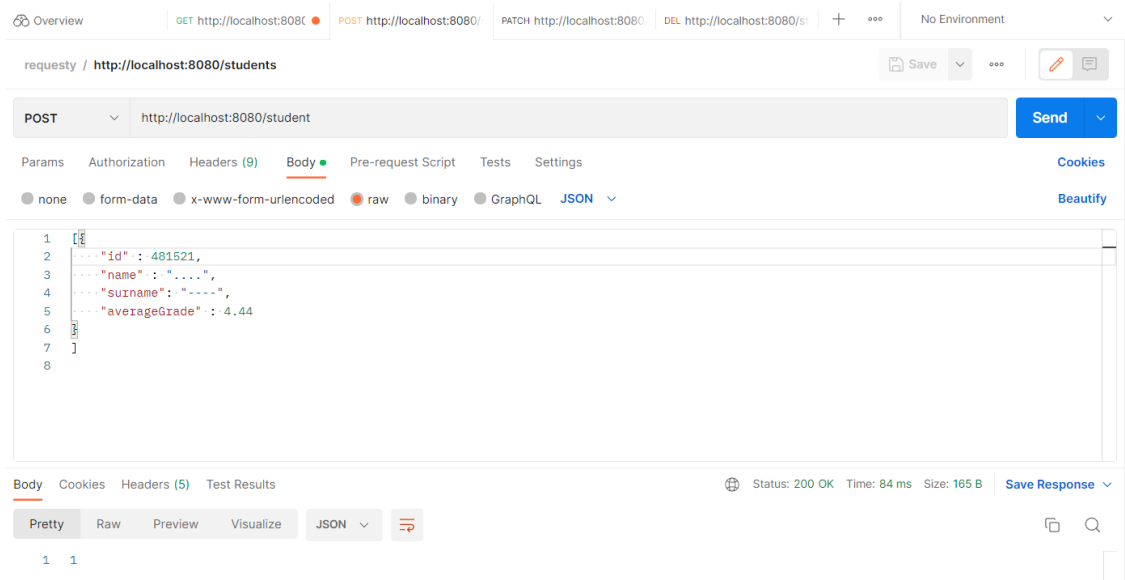


```
        students.forEach(student ->
jdbcTemplate.update("INSERT INTO student(id, name, surname,
averageGrade) VALUES(?, ?, ?, ?)",
        student.getId(), student.getName(),
student.getSurname(), student.getAverageGrade()
        ));
        return 1;
    }

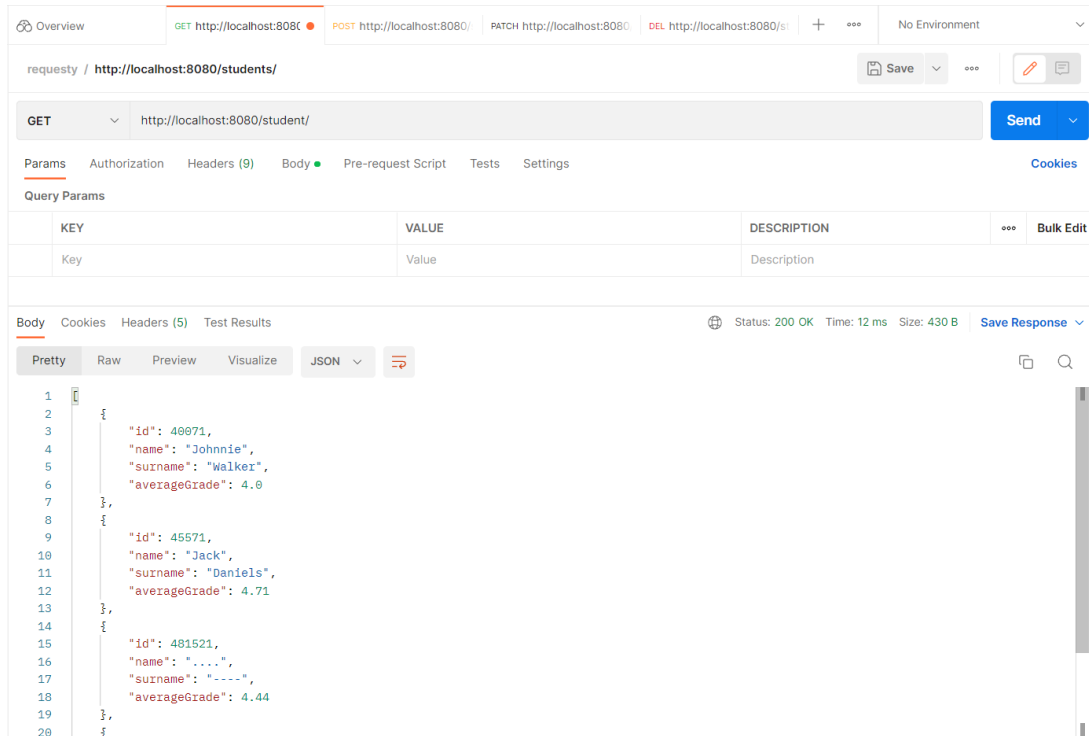
    public int update(Student student){
        return jdbcTemplate.update("UPDATE student SET name=?,
surname=?, averageGrade=? WHERE id=?",
        student.getName(), student.getSurname(),
student.getAverageGrade(), student.getId());
    }

    public int delete(int id){
        return jdbcTemplate.update("DELETE FROM student WHERE
id=?", id);
    }
    /*
    ...
    */
}
```

W celu sprawdzenia poprawności można posłużyć się przeglądarką (tylko dla żądań typu GET – `getAll()` lub `getById(id)`), dla pozostałych należy wykorzystać oprogramowanie pozwalające na przesyłanie żądania z treścią postaci np. JSON. W przykładzie zostało wykorzystane oprogramowanie Postman. Przykładowe żądania przedstawiono na rysunkach 8.11 – 8.12.



Rys. 8.11. Żądanie typu POST.



Rys. 8.12. Żądanie typu GET.

Do poprawnego uruchomienia aplikacji można wykorzystać przycisk Run (Shift + F10) lub polecenie terminala:

```
mvn spring-boot:run
```

Należy zdawać sobie sprawę z różnic między żądaniami typu PUT oraz PATCH, zwłaszcza przy generowaniu zapytań – nie jest to jednak celem laboratorium. Dla przypomnienia PUT służy do stworzenia nowego obiektu lub aktualizacji istniejącego, więc przy zapytaniu wymagany jest komplet informacji – wszystkie dane muszą być przekazane, ponieważ w przeciwnym razie utworzony/aktualizowany obiekt będzie miał pola z wartościami domyślnymi. Z kolei PATCH aktualizuje już istniejące obiekty, więc nie jest wymagany komplet danych podczas wysyłania żądania. Nie da się jednak stworzyć nowego obiektu.

Zadanie 8.1. Aplikacja do prowadzenia przychodni

Napisz aplikację do zarządzania bazą danych w przychodni. Aplikacja powinna zawierać zestaw metod do prowadzenia operacji typu CRUD:

- rejestracja pacjentów,
- aktualizacja danych,
- usuwanie wizyt,
- sprawdzanie wizyt (podgląd danych).

Zadanie 8.2. Aplikacja do prowadzenia restauracji

Napisz aplikację do zarządzania bazą danych w restauracji. Aplikacja powinna zawierać zestaw metod do prowadzenia operacji typu CRUD:

- dodawanie zamówień,
- aktualizacja zamówień,
- usuwanie zamówień,
- sprawdzanie zamówień.

LABORATORIUM 9. TWORZENIE APLIKACJI Z WYKORZYSTANIEM BIBLIOTEKI HIBERNATE.

Cel laboratorium:

Zapoznanie studentów z biblioteką Hibernate. Tworzenie aplikacji z wykorzystaniem biblioteki Hibernate.

Zakres tematyczny zajęć:

- Hibernate.
- HQL.
- Tworzenie projektu w oparciu o Hibernate.

Pytania kontrolne:

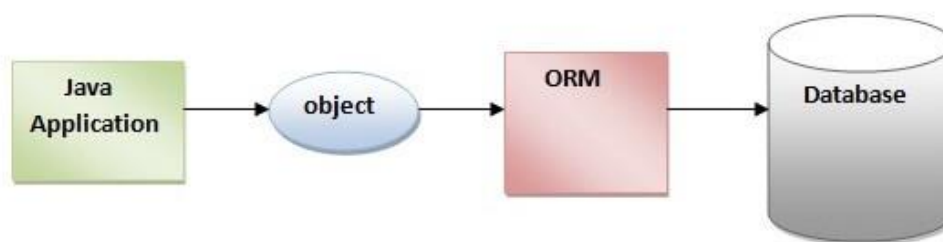
1. Czym jest Hibernate?
2. Rozszyfruj skrót ORM? Do czego jest wykorzystywane?
3. Co to jest HQL?
4. Do jakich celów wykorzystywany jest obiekt klasy SessionFactory?
5. Czy Hibernate wspiera adnotacje?
6. Podaj przykłady adnotacji z wyjaśnieniem.

Hibernate

Hibernate to biblioteka Javy, który upraszcza tworzenie aplikacji współpracujące z bazami danych. Jest to lekkie, otwarte narzędzie ORM (ang. Object Relational Mapping). Hibernate implementuje specyfikację JPA (Java Persistence API) dla trwałości danych. ORM jest techniką programowania, która odwzorowuje obiekt na dane przechowywane w bazie danych. Dzięki takiemu mapowaniu dużo prostsze jest tworzenie danych, manipulacja nimi oraz dostęp. Narzędzie ORM wykorzystuje interfejs API JDBC do interakcji z bazą danych.

Podstawowe zalety stosowania biblioteki Hibernate:

- lekka biblioteka open source,
- pamięć podręczna (ang. cache) podzielona na poziomy (przy czym poziom pierwszy L1 jest zawsze włączony) zapewnia szybkość i wydajność,
- możliwe jest tworzenie zapytań niezależnie od bazy danych dzięki HQL (ang. Hibernate Query Language) – wersja SQL zorientowana obiektowo,
- automatyczne tworzenie tabel,
- ułatwione pobieranie danych z wielu tabel,
- statystyka i stan zapytań do bazy danych,
- pobieranie wybranych atrybutów obiektu bez konieczności ładowania całego obiektu.



Rys. 9.1. Schemat wykorzystania ORM (źródło: <https://www.javatpoint.com/images/hibernate/orm.jpg>).

Hibernate Query Language

Hibernate Query Language (HQL) jest zbudowany analogicznie do języka SQL (Structured Query Language), jednak nie zależy od tabeli bazy danych. Zamiast nazwy tabeli stosowana jest nazwa klasy w HQL. Jest to więc język zapytań niezależny od bazy danych. Podstawową różnicą jest to, że HQL jest dosłownie językiem zapytań - nie można za jego pomocą wstawiać, zmieniać ani usuwać danych - jedynie otrzymywać. Niezależność od bazy jest bardzo ciekawą cechą, zwłaszcza w sytuacji, gdy w istniejącym projekcie pojawia się konieczność zmiany bazy. Dzięki zastosowaniu Hibernate'a nie ma potrzeby rekonstrukcji metod generujących zapytania w przypadku zmiany bazy danych.

Interfejs Zapytań (ang. Query Interface) jest obiektową reprezentacją zapytania Hibernate'a. Obiekt ten jest tworzony poprzez metodę `createQuery()`. W tabeli 9.1 przedstawiono wybrane metody interfejsu zapytań Hibernate'a (Query Interface).

Tabela 9.1. Wybrane metody z interfejsu Query.

Metoda	Opis
<code>public int executeUpdate()</code>	Służy do wykonania zapytania typu UPDATE lub DELETE
<code>public List list()</code>	Służy do wygenerowania listy relacji
<code>public Query setFirstResult(int rowno)</code>	Służy do ustawienia wiersza, z którego zostanie pobrany rekord
<code>public Query setMaxResult(int rowno)</code>	Służy do określenia liczby rekordów pobieranych z tabeli
<code>public Query setParameter(int position, Object value)</code>	Służy do ustawienia wartości parametru zgodnie ze składnią sterownika JDBC
<code>public String getQueryString()</code>	Służy do pobrania zapytania w postaci łańcucha znaków

Projekt oparty na Hibernate

Aplikacja wykorzystująca bibliotekę Hibernate powinna posiadać dodaną odpowiednią zależność (podobnie jak w innych projektach) oraz załadowany sterownik bazy danych. Ponad to niezbędne jest zdefiniowanie metody zwracającej obiekt klasy `SessionFactory`, dzięki któremu możliwe jest łączenie się z bazą danych. Podobnie jak w innych projektach potrzebny jest plik konfiguracyjny – w tym przypadku będzie to `hibernate.cfg.xml`, gdzie określone jest połączenie z bazą danych i klasy encji.

Dodanie zależności Hibernate w pliku pom.xml można zrealizować następująco:

```
<!--  
https://mvnrepository.com/artifact/org.hibernate.orm/hibernate  
-core -->  
<dependency>  
  <groupId>org.hibernate.orm</groupId>  
  <artifactId>hibernate-core</artifactId>  
  <version>6.1.2.Final</version>  
</dependency>
```

Dodatkowo można jednocześnie zdefiniować jaki typ bazy danych będzie obsługiwany – tu jako przykład został podany MySQL:

```
<!-- https://mvnrepository.com/artifact/mysql/mysql-connector-  
java -->  
<dependency>  
  <groupId>mysql</groupId>  
  <artifactId>mysql-connector-java</artifactId>  
  <version>8.0.30</version>  
</dependency>
```

Kolejnym krokiem jest utworzenie pliku konfiguracyjnego zwyczajowo nazywany hibernate.cfg.xml – zawiera on informację dotyczące:

- nazwy sterownika bazy danych,
- nazwy bazy danych,
- hosta,
- użytkownika i hasła,

Na końcu należy umieścić informacje o klasach encji z pakietem. Plik można utworzyć ręcznie w katalogu resources lub automatycznie. W tabeli 9.2 zostały zebrane nazwy dla sterowników różnych baz danych.

Tabela 9.2. Nazwy sterowników baz danych.

Baza danych	Sterownik
MySQL	com.MySql.Jdbc.Driver
HSQLDB	org.Hsqldb.JdbcDriver
Sybase	com.Sybase.Jdbc3.Jdbc.SybDriver
Apache Derby	org.Apache.Derby.Jdbc.EmbeddedDriver
IBM DB2	com.Ibm.Db2.Jcc.DB2Driver
PostgreSQL	org.Postgresql.Driver
SQL Server (Microsoft Driver)	com.Microsoft.Sqlserver.Jdbc.SQLServerDriver
Informix	com.Informix.Jdbc.IfxDriver
H2	org.H2.Driver



```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC "-//
//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <property
name="hibernate.connection.driver_class">com.mysql.jdbc.Driver
</property>
        <property
name="hibernate.connection.url">jdbc:mysql://localhost:3306/db
_name</property>
        <property
name="hibernate.connection.username">user</property>
        <property
name="hibernate.connection.password">password</property>
        <property name="hibernate.show_sql">true</property>
        <property
name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</p
roperty>
        <mapping class="Student" />

    </session-factory>
</hibernate-configuration>
```

Definiowanie klasy encji jest bardzo proste – zawiera podstawowe pola oraz metody typu get i set, toString. Klasa Student definiująca pola: id, imię, nazwisko, średnią ocen jest typową klasą modelową i może wyglądać następująco:

```
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.Id;

@Entity
public class Student {

    @GeneratedValue
    @Id
    private int id;
    private String name;
    private String surname;
    private double averageGrade;

    public Student() {
    }

    public Student(int id, String name, String surname, double
averageGrade) {
        this.id = id;
    }
}
```



```
        this.name = name;
        this.surname = surname;
        this.averageGrade = averageGrade;
    }

    @Override
    public String toString() {
        return "Student{" +
            "id = " + id +
            ", name = " + name +
            ", surname = " + surname +
            ", averageGrade = " + averageGrade +
            "}";
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getSurname() {
        return surname;
    }

    public void setSurname(String surname) {
        this.surname = surname;
    }

    public double getAverageGrade() {
        return averageGrade;
    }

    public void setAverageGrade(double averageGrade) {
        this.averageGrade = averageGrade;
    }
}
```



Domyślnie klasa zwracająca obiekt `SessionFactory` odpowiedzialny za tworzenie łącznie się z bazą może wyglądać następująco:

```
import org.hibernate.SessionFactory;
import
org.hibernate.boot.registry.StandardServiceRegistryBuilder;
import org.hibernate.cfg.Configuration;

public class HibernateFactory {
    public SessionFactory getSessionFactory() {
        Configuration configuration = new
Configuration().configure();
        StandardServiceRegistryBuilder registryBuilder =
            new StandardServiceRegistryBuilder()

        .applySettings(configuration.getProperties());
        SessionFactory sessionFactory = configuration
            .buildSessionFactory(registryBuilder.build());
        return sessionFactory;
    }
}
```

W tabeli 9.3 przedstawiono wybrane adnotacje biblioteki Hibernate.

Tabela 9.3. Wybrane adnotacje Hibernate'a.

Adnotacja	Opis
@Entity	służy do określenia klasy, która ma być mapowana – klasa powinna być zdefiniowana jako model danych (prywatne pola, publiczne metody typu get i set)
@GeneratedValue	służy do określenia sposobu generowania wartości kluczy podstawowych (bez parametrów – auto-generowanie)
@Id	służy do określenia klucza podstawowego. Może być stosowana do pola, którego typ jest podstawowy, opakowany (np. Integer) lub należy do jednej z klas: String, java.util.Date, java.sql.Date, java.math.BigDecimal, java.math.BigInteger
@Table	określa tabelę podstawową dla encji. Kolejne tabele mogą być określone przy pomocy adnotacji @SecondaryTable lub @SecondaryTables
@Column	służy do określania mapowanej kolumny dla pola. Jeżeli nie jest określona obowiązują wartości domyślne
@Enumerated	określa pole jako typ wyliczeniowy. Może być używana w połączeniu z adnotacją @Basic lub @ElementCollection dla typów podstawowych lub kolekcji

Zadanie 9.1. Aplikacja do prowadzenia szkoły

Napisz aplikację do zarządzania bazą danych uczniów w szkole. Aplikacja powinna zawierać zestaw metod do prowadzenia operacji typu CRUD:

- dodawanie uczniów,



Fundusze Europejskie
Wiedza Edukacja Rozwój



**Rzeczpospolita
Polska**

Unia Europejska
Europejski Fundusz Społeczny



- aktualizacja danych,
- usuwanie uczniów,
- podgląd danych uczniów.

Dodatkowo zadbaj o możliwość przypisania uczniów do klasy.

Zadanie 9.2. Aplikacja do prowadzenia szkoły

Rozwiń aplikację z zadania 9.1 o możliwość zarządzania danymi nauczycieli. Przygotuj schemat ERD. Zadbaj o poprawne relacje na linii nauczyciel – klasa – uczeń. W jaki sposób można rozwiązać problem z tymczasowym przypisaniem nauczyciela do klasy (zastępstwo)?

LABORATORIUM 10. TWORZENIE APLIKACJI Z WYKORZYSTANIEM BIBLIOTEKI JSOUP.

Cel laboratorium:

Zapoznanie studentów z biblioteką Jsoup. Tworzenie aplikacji z wykorzystaniem biblioteki Jsoup.

Zakres tematyczny zajęć:

- Jsoup.
- Przetwarzanie HTML.

Pytania kontrolne:

1. W jaki sposób biblioteka Jsoup pozwala na operowanie na HTML'u?
2. Jakie znasz funkcje udostępnione przez bibliotekę?
3. Czy Jsoup wspiera elementy języka JavaScript?
4. Czy Jsoup może być wykorzystana do budowania plików XML?
5. W jaki sposób nawigować po pliku HTML?

Wstęp

Jsoup to biblioteka Java do pracy z rzeczywistym kodem HTML. Zapewnia bardzo wygodny interfejs do pobierania adresów URL oraz wydobywania i manipulowania danymi przy użyciu najlepszych metod HTML5 DOM i selektorów CSS. Jsoup implementuje specyfikację WHATWG HTML5 i analizuje HTML do tego samego DOM, co nowoczesne przeglądarki. Jsoup to biblioteka typu open-source, jest wciąż rozwijana, posiada dobrą dokumentację oraz płynne i elastyczne API. Biblioteka Jsoup może być również używana do analizowania i budowania XML. Podstawowe funkcje udostępniane przez bibliotekę to m.in.:

- ładowanie – pobieranie i parsowanie kodu HTML do obiektu klasy Document,
- filtrowanie – wybieranie żądanych danych jako obiekty klasy Elements i dalsze ich przetwarzanie,
- wyodrębnianie – pozyskiwanie atrybutów, tekstu i HTML węzłów,
- modyfikowanie – dodawanie/edycja/usuwanie węzłów i edycja ich atrybutów.

Dodanie zależności do projektu jest podstawą dodania biblioteki do projektu. Tradycyjnie należy znaleźć w repozytorium Maven'a odpowiednią pozycję i dodać ją do pliku pom.xml:

```
<!-- https://mvnrepository.com/artifact/org.jsoup/jsoup -->
<dependency>
  <groupId>org.jsoup</groupId>
  <artifactId>jsoup</artifactId>
  <version>1.15.3</version>
</dependency>
```



Fundusze Europejskie
Wiedza Edukacja Rozwój



**Rzeczpospolita
Polska**

Unia Europejska
Europejski Fundusz Społeczny



Jsoup ładuje kod HTML strony i buduje odpowiednie drzewo DOM (ang. Document Object Model). To drzewo działa w taki sam sposób jak DOM w przeglądarce, oferując metody podobne dostępnym w bibliotece jQuery i Vanilla języka JavaScript. Metody te służą do wybierania, przechodzenia, manipulowania tekstem/HTML/atributami oraz dodawania/usuwania elementów.

Do wypisania samego akapitu strony dzięki jsoup wystarczy odpowiednio przygotowana metoda select.

```
Document doc = Jsoup.connect("http://example.com").get();
doc.select("p").forEach(System.out::println);
```

Istotną kwestią jest to, że narzędzia dostępne w bibliotece jsoup interpretują tylko HTML — nie są brane pod uwagę elementy języka JavaScript. Dlatego zmiany w DOM, które normalnie miałyby miejsce po załadowaniu strony w przeglądarce obsługującej JavaScript, nie będą widoczne w jsoup.

Faza ładowania obejmuje pobieranie i analizowanie kodu HTML do obiektu klasy Document. Jsoup gwarantuje parsowanie dowolnego kodu HTML, od najbardziej niepoprawnego do całkowicie zweryfikowanego, tak jak zrobiłaby to nowoczesna przeglądarka. Można to osiągnąć, ładując String, InputStream, File lub URL. W poniższym przykładzie wykorzystano adres URL: <https://cs.pollub.pl>

```
String blogUrl = "https://cs.pollub.pl";
Document doc = Jsoup.connect(blogUrl).get();
```

Ze względu na rozciągłość dokumentu nie jest on przedstawiony w instrukcji. Spróbuj wypisać w konsoli zawartość obiektu doc.

Zwróć uwagę na metodę get, która reprezentuje wywołanie HTTP GET. Możesz również wykonać HTTP POST za pomocą metody post (lub możesz użyć metody, która otrzymuje typ metody HTTP jako parametr).

W przypadku wykrywania nieprawidłowych stanów można posłużyć się kodami stanów nieprawidłowych oraz konstrukcją try – catch (np. dla kodu 404), powinno się wyłapać wyjątek HttpStatusException:

```
try {
    Document doc404 = Jsoup.connect("https://spring.io/will-not-be-found").get();
} catch (HttpStatusException ex) {
    //...
}
```

Czasami połączenie musi być nieco bardziej dostosowane. Jsoup.connect(...) zwraca połączenie, które pozwala ustawić między innymi przeglądarkę użytkownika, limit czasu połączenia, pliki cookie i nagłówki:



```
Connection connection = Jsoup.connect(blogUrl);
connection.userAgent("Opera");
connection.timeout(2000);
connection.cookie("cookieName", "val123");
connection.cookie("cookieName", "val123");
connection.referrer("http://google.com");
connection.header("headersecurity", "xyz123");
Document docCustomConn = connection.get();
```

Ponieważ połączenie odbywa się za pomocą interfejsu, możliwe jest połączenie tych metod przed wywołaniem żądanej metody HTTP, na kształt buildera:

```
Document docCustomConn = Jsoup.connect(blogUrl)
    .userAgent("Opera")
    .timeout(2000)
    .cookie("cookieName", "val123")
    .cookie("cookieName", "val123")
    .referrer("http://google.com")
    .header("headersecurity", "xyz123")
    .get();
```

Teraz, gdy zostało dokonane przekonwertowanie kodu HTML na dokument, nadszedł czas, aby go nawigować i znaleźć to, czego szukamy. Tutaj podobieństwo do jQuery/JavaScript jest bardziej widoczne, ponieważ jego selektory i metody przechodzenia są podobne.

Metoda `select()` z klasy `Document` przyjmuje ciąg reprezentujący selektor, używając tej samej składni selektor jak w przypadku CSS lub JavaScript. Następnie pobiera pasującą listę elementów. Ta lista może być pusta, ale nie może nie mieć wartości (`null`).

```
Elements links = doc.select("a");
Elements sections = doc.select("section");
Elements logo = doc.select(".spring-logo--container");
Elements pagination = doc.select("#pagination_control");
Elements divsDescendant = doc.select("header div");
Elements divsDirect = doc.select("header > div");
```

Przechodzenie po drzewie DOM (ang. *traversing*) jest również wspierane przez bibliotekę Jsoup. Zapewnia ona metody, które działają na dokumencie, na zestawie elementów lub na określonym elemencie, umożliwiając nawigację do rodziców, rodzeństwa lub dzieci węzła.

```
Element firstSection = sections.first();
Element lastSection = sections.last();
Element secondSection = sections.get(2);
Elements allParents = firstSection.parents();
Element parent = firstSection.parent();
Elements children = firstSection.children();
Elements siblings = firstSection.siblingElements();
```



Analogicznie możliwe jest przechodzenie po obiektach klasy Element za pomocą pętli np. for – each.

```
sections.forEach(el -> System.out.println("section: " + el));
```

W celu uzyskania dostępu do zawartości określonych elementów (np. atrybuty, tekst podrzędny lub kod HTML) można wykorzystać metody select(), attr(), text() oraz html().

```
Element firstArticle = doc.select("article").first();
Element timeElement = firstArticle.select("time").first();
String dateTimeOfFirstArticle = timeElement.attr("datetime");
Element sectionDiv = firstArticle.select("section
div").first();
String sectionDivText = sectionDiv.text();
String articleHtml = firstArticle.html();
String outerHtml = firstArticle.outerHtml();
```

Zadanie 10.1. Dostęp do treści

Napisz aplikację do nawigowania i odczytywania treści ze strony <https://cs.pollub.pl>. Pobierz aktualną listę pracowników z podziałem na zakłady. Lista pracowników powinna być posortowana alfabetycznie w ramach każdego zakładu.

Zadanie 10.2. Filtrowanie

Bazując na aplikacji z zadania 10.1 dodaj funkcjonalność wydzielającą z uzyskanej listy osoby posiadające stopień dr lub dr inż.

Zadanie 10.3. Wyszukiwanie

Do aplikacji dodaj funkcjonalność wyszukiwania informacji o godzinach dziekańskich lub rektorskich. Wyniki dodaj do listy w kolejności od najstarszego do najmłodszego wydarzenia (względem dat).



LABORATORIUM 11. WYBRANE ZAGADNIENIA DOTYCZĄCE BEZPIECZEŃSTWA.

Cel laboratorium:

Zapoznanie studentów z wybranymi zagadnieniami dotyczącymi bezpieczeństwa.

Zakres tematyczny zajęć:

- Podstawy kryptografii w Javie.
- Wybrane klasy pakietu security.
- Uwierzytelnienie.

Pytania kontrolne:

1. Jakie znasz zagrożenia na poziomie języka?
2. Jakie znasz algorytmy szyfrujące?
3. W jaki sposób bezpiecznie przechować dane użytkownika (np. login i hasło)?
4. Jak działa podpis pliku XML?

Wstęp

Bezpieczeństwo to bardzo obszerny temat, który obejmuje wiele obszarów. Niektóre z nich składają się na sam język. Przykładem takich elementów mogą być modyfikatory dostępu. Poza tym inne mogą być dostępne jako usługi, które obejmują między innymi szyfrowanie danych, uwierzytelnianie, bezpieczną komunikację oraz autoryzację. Jak widać temat jest bardzo rozległy i w tej części zostaną przedstawione tylko wybrane elementy dla lepszego poznania i praktycznego tworzenia bezpiecznych aplikacji.

Podstawowe bezpieczeństwo zaczyna się już na poziomie języka. Java jest językiem posiadającym wiele ukrytych na pierwszy rzut oka cech bezpieczeństwa. Wspomniane modyfikatory dostępu pozwalają na rozdział pól, metod i klas w taki sposób, aby dostęp do nich był ściśle kontrolowany. Kolejnym aspektem jest zarządzanie pamięcią, które w języku Java odbywa się automatycznie. Programista nie musi się kłopotać prozaicznymi czynnościami jak np. usuwanie obiektów ponieważ zajmuje się tym tzw. Garbage Collector. Statyczne typowanie danych pozwala zmniejszyć ilość błędów, które mogą przemknąć się przez proces kompilacji i ukazać dopiero podczas uruchomienia aplikacji. Wreszcie kod bajtowy po kompilacji jest weryfikowany i może działać niezależnie od platformy.

Kryptografia jest podstawą funkcji bezpieczeństwa w Javie. Obszar ten dotyczy zarówno narzędzi jak i technik bezpiecznej komunikacji w obecności zagrożeń. Kryptograficzna Architektura Java (ang. Java Cryptographic Architecture – JCA) zapewnia strukturyzowany dostęp i implementację funkcji kryptograficznych w tym języku. Należą do nich:

- podpisy cyfrowe,
- szyfrowanie symetryczne i asymetryczne,
- kody uwierzytelniania wiadomości,
- generatory kodów/kluczy oraz fabryki kluczy,

Java zawiera również wbudowanych dostawców szeroko używanych algorytmów kryptograficznych ale ważniejsze jest to, że wykorzystywane są implementacje oparte na dostawcy (tzw. Provider-based implementation) dla funkcji kryptograficznych. Przykładowe algorytmy należące do podstawowej Javy to m.in.: RSA, DSA, AES.

Podstawowym przypadkiem użycia kryptografii jest przechowywanie haseł użytkowników. Przechowywanie haseł jest niezbędne w celu późniejszego uwierzytelnienia użytkownika podczas np. próby logowania w celu nadania uprawnień na wyższym poziomie. Podczas pisania aplikacji bardzo często zdarza się, że hasło jest przechowywane jako zwykły łańcuch znaków. Na potrzeby nauki tworzenia przykładowej aplikacji GUI z ekranem logowania może być to wystarczające. Oczywistym jest jednak fakt, że nie jest to w żaden sposób bezpieczne. Jednym z rozwiązań jest zastosowanie kryptograficznej funkcji skrótu. Popularnym algorytmem do przeprowadzenia takiej operacji jest SHA-1 (Secure Hash Algorithm 1). Polega on na szyfrowaniu tylko w jedną stronę. Inaczej jest to kryptograficzna funkcja skrótu, która pobiera dane wejściowe i generuje 160-bitową (20-bajtową) wartość skrótu. Ta wartość skrótu jest znana jako skrót wiadomości. Ten skrót wiadomości jest zwykle renderowany jako liczba szesnastkowa o długości 40 cyfr. Przykładowy kod został przedstawiony poniżej:

```
import java.math.BigInteger;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.util.ArrayList;
import java.util.List;

public class Main {

    public static void main(String[] args) throws
NoSuchAlgorithmException {

        String[] pwds = {"pass","pass","password", "haslo",
"haslo123", "passpasspasspassword"};

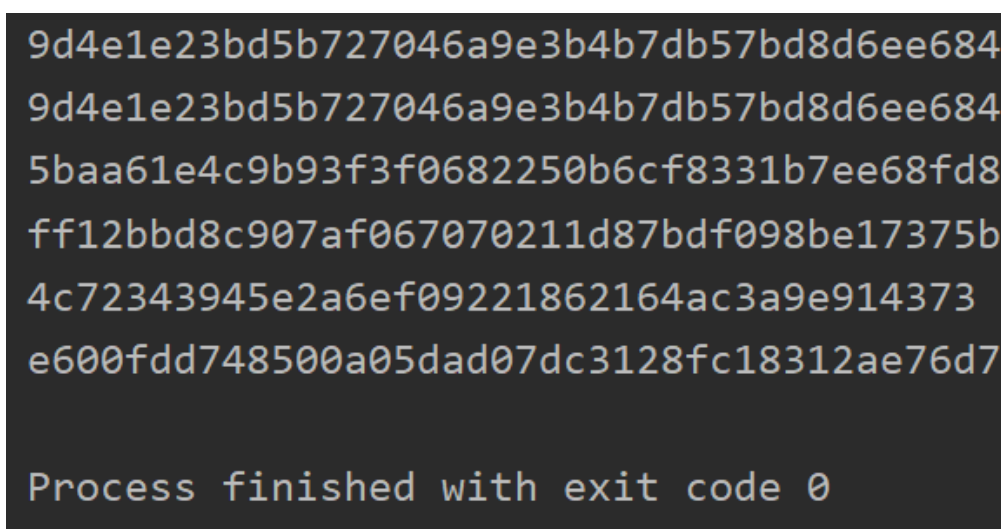
        List<String> afterAll = encrypt(pwds);
        for (String s:afterAll) {
            System.out.println(s);
        }
    }

    public static List<String> encrypt(String[] input) throws
NoSuchAlgorithmException {
        List<String> output = new ArrayList<>();
        MessageDigest messageDigest =
MessageDigest.getInstance("SHA-1");
        for (String s : input) {
            byte[] afterSHA =
messageDigest.digest(s.getBytes());
```




```
        BigInteger bigInteger = new BigInteger(1,  
afterSHA);  
        String hashtext = bigInteger.toString(16);  
        while (hashtext.length() < 32) {  
            hashtext = "0" + hashtext;  
        }  
        output.add(hashtext);  
    }  
    return output;  
}  
}
```

Metoda encrypt pobiera tablicę Stringów i zwraca listę po zastosowaniu funkcji szyfrującej opartej o algorytm SHA-1. Każdorazowo kolejny łańcuch znaków z tablicy wejściowej na podstawie wspomnianego algorytmu jest zamieniany na tablicę bajtów, konwertowany do reprezentacji signum, a następnie do formatu szesnastkowego. Po konwersjach łańcuch trafia do listy wyjściowej. Dla tak napisanego fragmentu istotna z punktu widzenia zabezpieczeń jest klasa MessageDigest, która jest usługą kryptograficzną. Stosując metodę getInstance() wysyłamy żądanie dla dostawcy zabezpieczeń. Wyniki otrzymane po uruchomieniu tej prostej aplikacji zostały przedstawione na rysunku 11.1.



```
9d4e1e23bd5b727046a9e3b4b7db57bd8d6ee684  
9d4e1e23bd5b727046a9e3b4b7db57bd8d6ee684  
5baa61e4c9b93f3f0682250b6cf8331b7ee68fd8  
ff12bbd8c907af067070211d87bdf098be17375b  
4c72343945e2a6ef09221862164ac3a9e914373  
e600fdd748500a05dad07dc3128fc18312ae76d7  
  
Process finished with exit code 0
```

Rys. 11.1. Funkcja skrótu dla poniższej tablicy.

Dla każdego łańcucha znaków należących do tablicy wejściowej:

```
{"pass", "pass", "password", "haslo", "haslo123",  
"passpasspasspassword"}
```

został wygenerowany 40 znakowy skrót składający się z liczb szesnastkowych. Warto zauważyć, że stosując dany algorytm z takiego samego łańcucha otrzymamy taki sam wynik. Widać to wyraźnie w dwóch pierwszych liniach, gdzie jako wejście został podany taki sam łańcuch znaków. Drugą istotną kwestią jest przełożenie długości wejściowej na wyjście – jak widać długość skrótu jest zawsze stała. Trzecia uwaga dotyczy zawierania się jednego



łańcucha w drugim – np. pierwszy i ostatni element tablicy. Nie ma to żadnego znaczenia, ponieważ patrz poprzedni punkt.

Inne algorytmy, które mogą być obsługane przez klasę MessageDigest to m.in.:

- MD2
- MD5
- SHA-1
- SHA-224
- SHA-256
- SHA-384
- SHA-512

Klucz publiczny

Infrastruktura klucza publicznego (PKI) odnosi się do konfiguracji, która umożliwia bezpieczną wymianę informacji w sieci przy użyciu szyfrowania klucza publicznego. Ta konfiguracja opiera się na zaufaniu budowanemu między stronami zaangażowanymi w komunikację. Zaufanie opiera się na certyfikatach cyfrowych wydanych przez neutralny i zaufany urząd, znany jako urząd certyfikacji (CA).

Platforma Java posiada interfejsy API ułatwiające tworzenie, przechowywanie i walidację certyfikatów cyfrowych. Do podstawowych należą:

- KeyStore,
- CertStore.

Java udostępnia klasę KeyStore do trwałego przechowywania kluczy kryptograficznych i zaufanych certyfikatów. W tym przypadku KeyStore może reprezentować zarówno pliki magazynu kluczy, jak i pliki magazynu zaufania. Pliki te mają podobną zawartość, ale różnią się pod względem wykorzystania. Kolejną klasą w Javie jest CertStore, która reprezentuje publiczne repozytorium potencjalnie niezaufanych certyfikatów i list odwołań. Certyfikaty i listy odwołań powinny być pobrane w celu utworzenia ścieżki certyfikatu. Java ma wbudowany magazyn zaufania o nazwie „cacerts”, który zawiera certyfikaty dla dobrze znanych urzędów certyfikacji.

Kolejnym przydatnym narzędziem ułatwiającym komunikację opartą o zaufanie są:

- keytool,
- jarsigner.

Narzędzie o nazwie „keytool” służy do tworzenia i zarządzania magazynem kluczy i magazynem zaufania. Kolejne narzędzie „jarsigner”, służy do podpisywania i weryfikacji plików JAR.

Do nawiązania bezpiecznego połączenia za pomocą SSL w języku Java potrzebne są dwie rzeczy: Present Certificate oraz Verify Certificate.

Present Certificate – w komunikacji musimy przedstawić ważny certyfikat innej stronie. W tym celu musimy załadować plik magazynu kluczy (keyStore), w którym musimy mieć nasze klucze publiczne:

```
//path to keystore.jks...
public void test(String path) throws KeyStoreException {
    KeyStore keyStore =
    KeyStore.getInstance(KeyStore.getDefaultType());
```



```
char[] keyStorePassword = "tryMeBro".toCharArray();
try(InputStream keyStoreData = new
FileInputStream(path)) {
    keyStore.load(keyStoreData, keyStorePassword);
} catch (FileNotFoundException e) {
    e.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
} catch (CertificateException e) {
    e.printStackTrace();
} catch (NoSuchAlgorithmException e) {
    e.printStackTrace();
}
}
```

Verify Certificate – należy również zweryfikować certyfikat przedstawiony przez drugą stronę w komunikacji. W tym celu należy załadować trustStore, w którym wcześniej musimy mieć zaufane certyfikaty od innych stron:

```
//path to truststore.jks...
public void checkMe(String path) throws KeyStoreException {
    KeyStore trustStore =
KeyStore.getInstance(KeyStore.getDefaultType());
    /*
    char[] trustStorePassword = ...
    */
    try(InputStream trustStoreData = new
FileInputStream(path)) {
        trustStore.load(trustStoreData,
trustStorePassword);
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    } catch (CertificateException e) {
        e.printStackTrace();
    } catch (NoSuchAlgorithmException e) {
        e.printStackTrace();
    }
}
}
```

Można wykonywać to programowo ale bardzo często zdarza się, że przekazanie parametrów następuje już w czasie wykonywania aplikacji poprzez parametry systemowe:

```
-Djavax.net.ssl.trustStore=truststore.jks
-Djavax.net.ssl.keyStore=keystore.jks
```

Uwierzytelnienie

Uwierzytelnienie jest to proces weryfikacji (potwierdzenia) tożsamości użytkownika lub hosta, na podstawie danych uwierzytelniających np. hasło, token, plik.

Java API wykorzystuje moduły logowania oparte na pluginach, aby zapewnić aplikacjom różne i często wielorakie mechanizmy uwierzytelniania. LoginContext udostępnia abstrakcyjny poziom uwierzytelniania, natomiast za konfigurację i ładowanie modułów odpowiada LoginModule. Wiele rodzajów dostawców udostępnia moduły do logowania, jednak Java sama w sobie również pozwala na użycie domyślnych modułów takich jak:

- Krb5LoginModule – do uwierzytelniania opartego na protokole Kerberos,
- JndiLoginModule – do uwierzytelniania opartego na nazwie użytkownika i hasle wspieranym przez magazyn LDAP,
- KeyStoreLoginModule – do uwierzytelniania opartego na kluczu kryptograficznym.

Typowym mechanizmem uwierzytelniania jest nazwa użytkownik (login) oraz hasło. Wykorzystując jeden z powyższych modułów (JndiLoginModule) możliwe jest nie tylko pobranie loginu oraz hasła ale również weryfikacja ich z usługą katalogową skonfigurowaną zgodnie z JNDI ([źródło](#)).

```
package org.jboss.book.security.ex2;

import java.security.acl.Group;
import java.util.Map;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.security.auth.Subject;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.login.LoginException;

import org.jboss.security.SimpleGroup;
import org.jboss.security.SimplePrincipal;
import
org.jboss.security.auth.spi.UsernamePasswordLoginModule;

/**
 * An example custom login module that obtains passwords and
roles
 * for a user from a JNDI lookup.
 *
 * @author Scott.Stark@jboss.org
 * @version $Revision: 1.12 $
 */
public class JndiUserAndPass
    extends UsernamePasswordLoginModule
{
    /** The JNDI name to the context that handles the
password/username lookup */
    private String userPathPrefix;
```



```
/** The JNDI name to the context that handles the roles/  
username lookup */  
private String rolesPathPrefix;  
  
/**  
 * Override to obtain the userPathPrefix and  
rolesPathPrefix options.  
 */  
public void initialize(Subject subject, CallbackHandler  
callbackHandler,  
                        Map sharedState, Map options)  
{  
    super.initialize(subject, callbackHandler,  
sharedState, options);  
    userPathPrefix = (String)  
options.get("userPathPrefix");  
    rolesPathPrefix = (String)  
options.get("rolesPathPrefix");  
}  
  
/**  
 * Get the roles the current user belongs to by querying  
the  
 * rolesPathPrefix + '/' + super.getUsername() JNDI  
location.  
 */  
protected Group[] getRoleSets() throws LoginException  
{  
    try {  
        InitialContext ctx = new InitialContext();  
        String rolesPath = rolesPathPrefix + '/' +  
super.getUsername();  
  
        String[] roles = (String[]) ctx.lookup(rolesPath);  
        Group[] groups = {new SimpleGroup("Roles")};  
        log.info("Getting roles for  
user="+super.getUsername());  
        for(int r = 0; r < roles.length; r++) {  
            SimplePrincipal role = new  
SimplePrincipal(roles[r]);  
            log.info("Found role="+roles[r]);  
            groups[0].addMember(role);  
        }  
        return groups;  
    } catch(NamingException e) {  
        log.error("Failed to obtain groups for  
user="+super.getUsername(), e);  
        throw new LoginException(e.toString(true));  
    }  
}
```



```
    }

    /**
     * Get the password of the current user by querying the
     * userPathPrefix + '/' + super.getUsername() JNDI
location.
     */
    protected String getUsersPassword()
        throws LoginException
    {
        try {
            InitialContext ctx = new InitialContext();
            String userPath = userPathPrefix + '/' +
super.getUsername();
            log.info("Getting password for
user="+super.getUsername());
            String passwd = (String) ctx.lookup(userPath);
            log.info("Found password="+passwd);
            return passwd;
        } catch (NamingException e) {
            log.error("Failed to obtain password for
                        user="+super.getUsername(), e);
            throw new LoginException(e.toString(true));
        }
    }
}
```

Kontrola dostępu

Kontrola dostępu odnosi się do ochrony wrażliwych zasobów, takich jak system plików lub baza plików źródłowych przed nieuprawnionym dostępem. Zazwyczaj osiąga się to poprzez ograniczenie dostępu do takich zasobów.

Możemy osiągnąć kontrolę dostępu w Javie za pomocą klas Policy i Permission za pośrednictwem klasy SecurityManager. SecurityManager jest częścią pakietu „java.lang” i jest odpowiedzialny za egzekwowanie kontroli dostępu w Javie. Gdy moduł ładujący klasy ładuje klasę w środowisku wykonawczym, automatycznie przyznaje pewne domyślne uprawnienia klasie zawartej w obiekcie Permission. Poza tymi domyślnymi uprawnieniami, możemy zwiększyć wpływ na klasę poprzez zasady bezpieczeństwa. Są one reprezentowane przez zasady klasowe. Podczas sekwencji wykonywania kodu, jeśli środowisko wykonawcze napotka żądanie chronionego zasobu, SecurityManager weryfikuje żądane Permission z zainstalowaną Policy poprzez stos wywołań. W związku z tym albo udziela uprawnień, albo zgłasza SecurityException.

Innym sposobem jest wykorzystanie tzw. Java Tools for Policy. Java ma domyślną implementację strategii, która odczytuje dane autoryzacji z pliku właściwości. Wpisy zasad w tych plikach zasad muszą mieć określony format. Java jest dostarczana z „policytool”, graficznym narzędziem do tworzenia plików Policy. Dosyć dokładnie jest to opisane na stronie firmy Oracle:

<https://docs.oracle.com/javase/8/docs/technotes/guides/security/PolicyGuide.html>



Fundusze Europejskie
Wiedza Edukacja Rozwój



**Rzeczpospolita
Polska**

Unia Europejska
Europejski Fundusz Społeczny



Podpisywanie plików XML

Podpis XML jest przydatny w zabezpieczaniu danych i zapewnianiu integralności danych. W3C przedstawia zalecenia dotyczące zarządzania podpisem XML. Możliwe jest użycie podpisu XML do zabezpieczenia danych dowolnego typu, takich jak dane binarne. Java API obsługuje generowanie i walidację podpisów XML zgodnie z zalecanymi wytycznymi. Java XML Digital Signature API jest zawarty w pakiecie `java.xml.crypto`. Sam podpis jest tylko dokumentem XML. Podpisy XML mogą mieć trzy typy:

- Odłączony (detached) – ten typ podpisu jest nad danymi, które są zewnętrzne w stosunku do elementu Signature,
- Otaczający (enveloping) – ten typ podpisu jest nad danymi wewnętrznymi elementu Signature
- Otoczony (enveloped) – ten typ podpisu znajduje się nad danymi zawierającymi sam element Signature

Java wspiera tworzenie i weryfikację wszystkich powyższych typów podpisów XML.

Możemy chcieć wygenerować dokument XML dla danych w celu późniejszego wysłania go. Odbiorca powinien móc zweryfikować jego integralność.

Do wygenerowania dokumentu XML możemy posłużyć się klasą `XMLSignatureFactory`. Dane dla których tworzony jest dokument XML znajdują się w pliku `data.xml`. `XMLSignature` wymaga obiektu klasy `SignedInfo` dla którego wyliczany jest podpis, ponadto potrzebuje obiektu przechowującego klucz podpisu i certyfikat. `XMLSignature` podpisuje dokument przy użyciu klucza prywatnego w postaci `DOMSignContext`. Jako rezultat dokument XML będzie zawierał element Signature, który może posłużyć do weryfikacji jego integralności.

```
XMLSignatureFactory xmlSignatureFactory =
XMLSignatureFactory.getInstance("DOM");
DocumentBuilderFactory documentBuilderFactory =
DocumentBuilderFactory.newInstance();
documentBuilderFactory.setNamespaceAware(true);

Document document = documentBuilderFactory
    .newDocumentBuilder().parse(new
FileInputStream("data.xml"));

DOMSignContext domSignContext = new DOMSignContext(
    keyEntry.getPrivateKey(), document.getDocumentElement());

XMLSignature xmlSignature =
xmlSignatureFactory.newXMLSignature(signedInfo, keyInfo);
xmlSignature.sign(domSignContext);
```

Zadanie 11.1. Dane użytkownika

Napisz aplikację do rejestracji użytkowników – pobierz odpowiednie dane i zapisz je do pliku. Upewnij się, że użytkownik podaje odpowiednio długi łańcuch znaków. Zadbaj



o różnorodność (np. brak powtarzających się po sobie znaków – login111 lub konieczność używania wielkich i małych liter, cyfr i znaków specjalnych).

Zadanie 11.2. Logowanie

Na podstawie zadania 11.1 spróbuj przeprowadzić proces logowania. Wykorzystując plik z danymi sprawdź, czy podawane przez użytkownika login i hasło są identyczne. Ustaw limit prób. W każdym przypadku wyświetl odpowiedni komunikat. W jaki sposób można zabezpieczyć plik z hasłami?

LABORATORIUM 12. TESTY JEDNOSTKOWE JUnit.

Cel laboratorium:

Zapoznanie studentów z celowością przeprowadzania testów jednostkowych przy wykorzystaniu biblioteki JUnit.

Zakres tematyczny zajęć:

- Biblioteka JUnit.
- Testy jednostkowe.

Pytania kontrolne:

1. Czym jest JUnit?
2. Co to jest test jednostkowy?
3. Po co należy testować aplikacje?
4. Jakie znasz inne rodzaje testów?
5. Czy napisanie testu gwarantuje poprawnie działającą aplikację?

JUnit

JUnit to jedna z najpopularniejszych bibliotek do testów jednostkowych w ekosystemie Java. Istnieje kilka wersji biblioteki – najnowsza jest JUnit 5. Aktualna wersja biblioteki zawiera szereg ciekawych innowacji, których celem jest wspieranie nowych funkcji od Javy 8 do nowszych wersji JDK, a także umożliwia prowadzenie testów na wiele różnych sposobów. Konfiguracja jest analogiczna jak w przypadku innych bibliotek, o ile projekt jest tworzony w oparciu o Maven'a. Wystarczy w pliku pom.xml dodać odpowiednią zależność i umożliwić pobranie odpowiednich plików. Zależność dla najnowszej wersji 5.9.0 została przedstawiona poniżej.

```
<!--  
https://mvnrepository.com/artifact/org.junit.jupiter/junit-  
jupiter-api -->  
<dependency>  
  <groupId>org.junit.jupiter</groupId>  
  <artifactId>junit-jupiter-api</artifactId>  
  <version>5.9.0</version>  
  <scope>test</scope>  
</dependency>
```

Dodatkowym wsparciem dla testów jednostkowych charakteryzują się niektóre środowiska programistyczne IDE takie jak np. IntelliJ IDEA lub Eclipse. Wówczas przeprowadzanie testów sprowadza się do odpowiedniego kliknięcia (w przypadku IntelliJ jest to PPM → Uruchom (Run)). Istotna jest wersja JDK – biblioteka wymaga wersji (co najmniej) Java 8.

JUnit można podzielić na kilka modułów. Pierwszym jest platforma JUnit. Służy do dostarczania odpowiednich poleceń testowych dla JVM. Jest więc to interfejs między



biblioteką JUnit, a narzędziami do budowania aplikacji. Kolejnym modulem jest JUnit Jupiter – jest to zestaw modeli programowania i rozszerzeń do tworzenia testów. Ostatnim modulem jest JUnit Vintage, który obejmuje testy na podstawie poprzednich wersji biblioteki JUnit 3 oraz 4 i pozwala je uruchomić na platformie w wersji 5. Pomocne adnotacje dostarczone przez moduł Jupiter zostały przedstawione poniżej:

- `@TestFactory` – oznacza metodę będącą fabryką testów dla testów dynamicznych,
- `@DisplayName` – definiuje niestandardową nazwę wyświetlaną dla klasy testowej lub metody testowej,
- `@Nested` – oznacza, że klasa z adnotacjami jest zagnieżdżoną, niestatyczną klasą testową,
- `@Tag` – deklaruje tagi do testów filtrujących,
- `@ExtendWith` – rejestruje niestandardowe rozszerzenia,
- `@BeforeEach` – oznacza, że metoda z adnotacjami zostanie wykonana przed każdą metodą testową (w poprzednich wersjach `@Before`),
- `@AfterEach` – oznacza, że metoda z adnotacjami zostanie wykonana po każdej metodzie testowej (w poprzednich wersjach `@After`),
- `@BeforeAll` – oznacza, że metoda z adnotacjami zostanie wykonana przed wszystkimi metodami testowymi w bieżącej klasie (w poprzednich wersjach `@BeforeClass`),
- `@AfterAll` – oznacza, że metoda z adnotacją zostanie wykonana po wykonaniu wszystkich metod testowych w bieżącej klasie (w poprzednich wersjach `@AfterClass`),
- `@Disable` – wyłącza klasę lub metodę testową (w poprzednich wersjach `@Ignore`).

Dzięki JUnit możliwe jest tworzenie powtarzalnych testów, które umożliwiają wykrycie jeszcze na etapie tworzenia aplikacji potencjalnych nieprawidłowości. Biblioteka zawiera szereg klas, ze względów oczywistych nie wszystkie będą omawiane.

Klasa `Assert` zawiera zestaw asercji pozwalających na porównanie wyników oczekiwanych z rzeczywistymi. Klasa `TestRunner` umożliwia wykonanie przypadku testowego. Klasa `TestCase` jako klasa abstrakcyjna pozwala na implementację metod `setUp()` oraz `tearDown()` do ustawienia danych wejściowych dla wykonania przypadku testowego oraz czyszczenie po przeprowadzeniu testu. `TestSuite` pozwala na grupowanie większej ilości testów.

Przykład trywialny od którego warto rozpocząć testowanie. Dana jest klasa zawierająca metodę, która ma zwracać liczbę: -1, 0 lub 1 w zależności od wejścia.

```
public class ToTest {

    public static int methodInt(int input){
        if (input > 0){
            return 1;
        }
        else if (input < 0){
            return -1;
        }else{
            return 0;
        }
    }
}
```



```
}
```

Testem jednostkowym będzie metoda testująca jednostkę, czyli w tym przypadku metodę z innej klasy. Adnotacja `@Test` pozwala na utworzenie testu. Metoda `methodInt` jest statyczna, więc nie jest niezbędne tworzenie instancji klasy `ToTest` (jeśli testujemy metodę klasową to należy przed asercją utworzyć instancję).

W katalogu test należy utworzyć klasę, w której zostanie utworzony test dla metody `methodInt`.

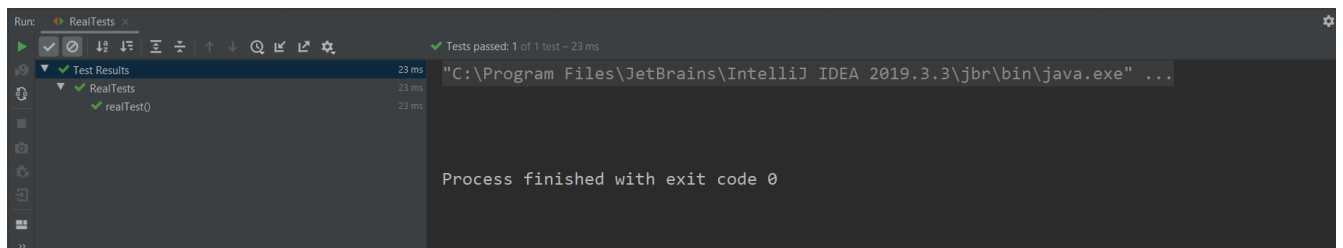
```
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.Assertions;

public class RealTests {

    @Test
    public void realTestOne(){
        Assertions.assertEquals(-1, ToTest.methodInt(-500));
        Assertions.assertEquals(0, ToTest.methodInt(0));
        Assertions.assertEquals(1, ToTest.methodInt(500));
    }

}
```

Po utworzeniu odpowiedniej metody testującej należy uruchomić ją – analogicznie jak uruchamiana jest metoda `main` w klasie głównej. Jeżeli metoda przechodzi test pojawiają się pożądanego zielone symbole – najczęściej tak jak na rysunku 12.1. Jeżeli sytuacja jest inna i z jakiegoś powodu metoda nie przechodzi testu to analogicznie kolor z zielonego zmienia się na pomarańczowy lub czerwony (rysunek 12.2).



Rys. 12.1. Przeprowadzenie testu jednostkowego.

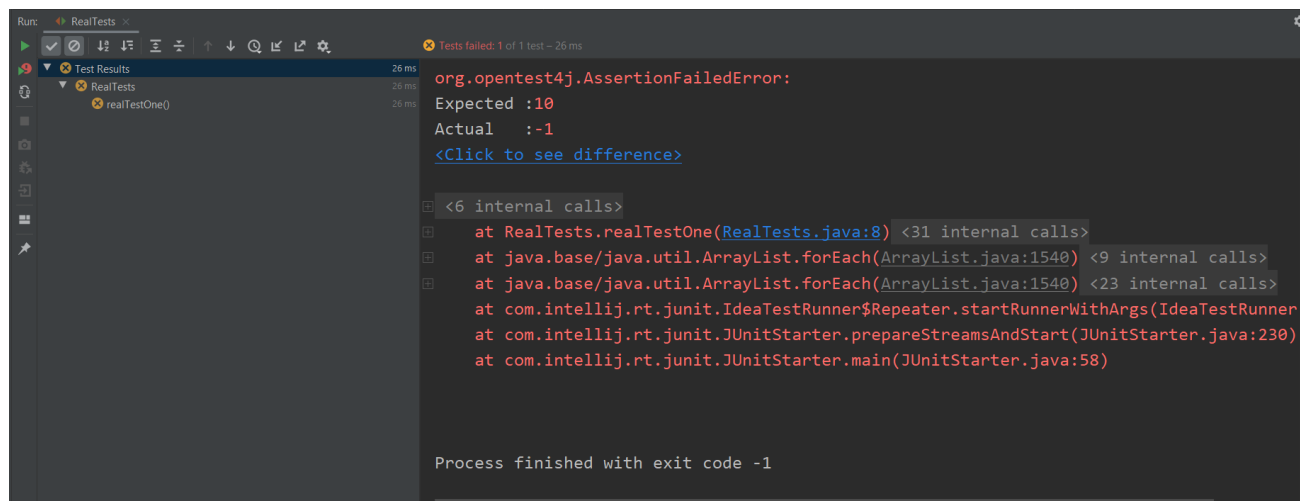
```
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.Assertions;

public class RealTests {

    @Test
    public void realTestOne(){
        Assertions.assertEquals(10, ToTest.methodInt(-500));
    }

}
```

```
        Assertions.assertEquals(10, ToTest.methodInt(0));
        Assertions.assertEquals(10, ToTest.methodInt(500));
    }
}
```



Rys. 12.2. Przeprowadzenie testu jednostkowego.

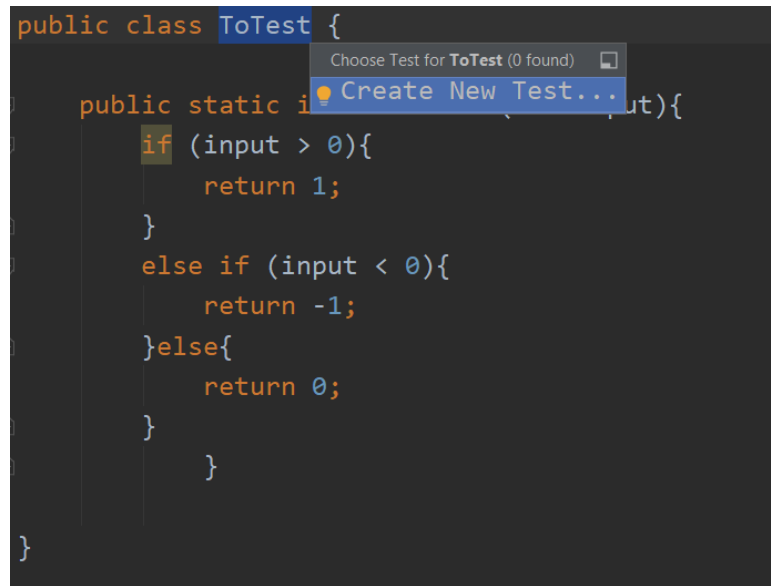
Jak widać bardzo dużo zależy od poprawnie napisanego testu. Dodatkowo należy zauważyć, że testy nie gwarantują zawsze poprawnie działającej aplikacji. Ich zadaniem jest zmniejszenie prawdopodobieństwa, że wadliwy kod trafi na dalsze etapy produkcji oprogramowania. Im więcej testów (poprawnie zaplanowanych i napisanych) zwiększa jakość kodu – o ile są brane pod uwagę.

JUnit dostarcza wiele metod – warto zapoznać się z poniższymi:

- `assertTrue` sprawdza czy przekazany argument to `true`,
- `assertFalse` sprawdza czy przekazany argument to `false`,
- `assertNull` sprawdza czy przekazany argument to `null`,
- `assertNotNull` sprawdza czy przekazany argument nie jest nullem,
- `assertEquals` przyjmuje dwa parametry wartość oczekiwaną i wartość rzeczywistą, jeśli są różne rzuca wyjątek,
- `assertNotEquals` przyjmuje dwa parametry wartość oczekiwaną i wartość rzeczywistą, rzuci wyjątek jeśli są równe.

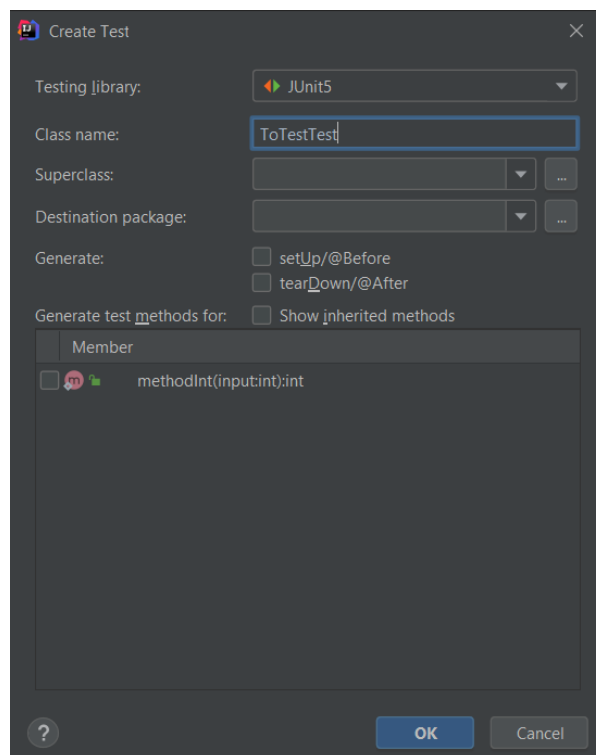
To tylko wycinek metod ale jak widać możliwości testowania jest bardzo wiele. Należy pamiętać, że warto pisać testy, które sprawdzają mały obszar – dużo łatwiej znaleźć błędny sposób działania. Testować należy przede wszystkim tzw. warunki brzegowe – książkowym przykładem jest metoda zwracająca dzielenie liczb całkowitych. Co należy wówczas przewidzieć?

W środowisku IntelliJ IDEA możliwe jest jeszcze prostsze tworzenie nowych klas testowych. Wystarczy zaznaczyć nazwę klasy do przetestowania i wykorzystać skrót klawiszowy `Ctrl + Shift + T` jak na rysunku 12.3.



Rys. 12.3. Tworzenie nowego testu.

Następnie należy utworzyć klasę z gotowymi metodami (zaznaczając przy nazwie, która metoda ma być testowana) lub wykorzystując później kombinację klawiszy Alt + Insert. Automatycznie klasa testowa otrzymuje nazwę bazowej klasy + suffix (Test).



Rys. 12.4. Tworzenie nowego testu.

Zadanie 12.1. Testy jednostkowe

Odszukaj w swoich zbiorach (lub utwórz nowe) aplikacje podstawowe, np. kalkulator, odbieranie wartości od użytkownika. Dla każdej metody napisz odpowiednią liczbę testów

(w zależności od tego co może się wydarzyć, a co nie powinno). Uruchom testy, czy należy coś zmienić w bazowej metodzie?

Zadanie 12.2. Testy jednostkowe

Napisz aplikację zawierającą zestaw metod do:

- odczytu wartości z pliku,
- sprawdzania czy wartości są liczbami całkowitymi,
- obliczania średniej na podstawie odczytanych liczb,
- zapisywania średniej do pliku.

Możesz posłużyć się metodami pomocniczymi. Co może pójść nie tak? Napisz testy jednostkowe dla każdej z tych metod, sprawdź wszystkie możliwe warianty.



Materiały zostały opracowane w ramach projektu
„*Zintegrowany Program Rozwoju Politechniki Lubelskiej – część druga*”,
umowa nr **POWR.03.05.00-00-Z060/18-00**
w ramach Programu Operacyjnego Wiedza Edukacja Rozwój 2014-2020
współfinansowanego ze środków Europejskiego Funduszu Społecznego